MySQL BASICS

The Foundation of Powerful Databases

RELATIONAL DATABASE

MySQL is a relational database that stores data in tables.

OPEN SOURCE

MySQL is open source and free to use under the GPL license.

SQL DATABASE

Uses SQL (Structured Query Language) for managing data.

TABLES

Data is organized in tables consisting of rows and columns.

ROWS

Rows represent individual records in a table.

COLUMNS

Columns represent the attributes or fields in a table.

FAST & RELIABLE

MySQL is known for its speed, performance and reliability.

EXAMPLE TABLE: *students*

COLUMNS

ROWS

id	name	email	age	course
1	Aman	aman@example.com	21	BCA
2	Priya	priya@example.com	20	B.Sc
3	Rahul	rahul@example.com	22	MCA

KEY POINTS

- Organize data in tables.
- Easy to use and manage.
- Supports powerful queries.
- Works on multiple platforms.
- Ideal for web & enterprise apps.

SECURE

Robust security features to protect your data.

SCALABLE

Handles small to large applications with ease.

FLEXIBLE

Supports a wide range of storage engines and features.

CROSS-PLATFORM

Runs on Windows, Linux, macOS and more.

HIGH PERFORMANCE

Optimized for speed, reliability and high availability.

MySQL INSTALLATION & SETUP

Easy Steps to Install and Connect to MySQL

1 DOWNLOAD MySQL

Download the official MySQL Installer from the Oracle website.

https://dev.mysql.com/downloads/

2 MySQL COMPONENTS

MySQL Server

The database server that stores and manages your data.

Required

MySQL Workbench

GUI tool for database designing, SQL development and administration.

Recommended

MySQL Shell

Advanced CLI for database management and scripting.

Optional

3 INSTALLATION STEPS

1. Run the MySQL Installer
2. Choose Custom or Developer Default
3. Select the components to install
4. Click Execute to install
5. Configure MySQL Server
6. Finish and launch MySQL Workbench

4 CONNECTION DETAILS

Host / Server	localhost
Port	3306
User	root
Authentication	Password

5 ROOT USER SETUP

The root user has full access to the MySQL Server.

Username	root
Default Host	localhost
Password	(set during installation)
Role	Administrator

TIP

Use a strong password and keep it secure.

6 CONNECT TO MySQL

1. Open MySQL Workbench (or MySQL Shell)
2. Enter Connection Details
   Host: localhost
   Port: 3306, User: root, Password
3. Click Test Connection
   If successful, you're ready to work with MySQL!

You're All Set!

MySQL is installed and ready to power your applications.

MySQL

QUICK INFO

- Default Port: 3306
- Default Host: localhost
- Default User: root
- Configuration File: my.ini (Windows) my.cnf (Linux)

Open Source | High Performance | Secure & Reliable | Cross-Platform | Easy to Use

DATABASE BASICS

The Building Blocks of MySQL Databases

1 DATABASE

A database is a collection of related data organized for efficient access, management and storage.

2 TABLE

A table is a collection of related data items consisting of rows and columns.

3 ROW

A row (or record) is a single entry in a table.

4 COLUMN

A column (or field) represents a specific attribute or property in a table.

EXAMPLE TABLE: *students*

COLUMNS

id	name	email	age	course
1	Aman	aman@example.com	21	BCA
2	Priya	priya@example.com	20	B.Sc
3	Rahul	rahul@example.com	22	MCA
4	Neha	neha@example.com	19	B.Com
5	Arjun	arjun@example.com	23	MBA

ROW

COLUMN

RECORD

5 RECORD

A record is a complete set of related data in a single row of a table.

6 SCHEMA

A schema is the structure that defines tables, columns, data types and relationships.

7 PRIMARY KEY

A primary key is a unique identifier for each record in a table. It ensures that no two rows have the same value in the primary key column.

Example

In our *students* table, 'id' is the Primary Key. It uniquely identifies each student.

8 RELATIONSHIPS

Relationships define how data in one table is related to data in another table. They help maintain data integrity.

Example

One-to-Many relationship between *students* and orders tables using *student_id*.

ORGANIZED DATA | DATA INTEGRITY | BETTER PERFORMANCE | EASY TO MANAGE | SCALABLE & RELIABLE

CREATE DATABASE

Manage and Organize Your Data with MySQL

1 CREATE DATABASE

Syntax:

```
CREATE DATABASE database_name;
```

2 USE DATABASE

Syntax:

```
USE database_name;
```

3 SHOW DATABASES

Syntax:

```
SHOW DATABASES;
```

4 DROP DATABASE

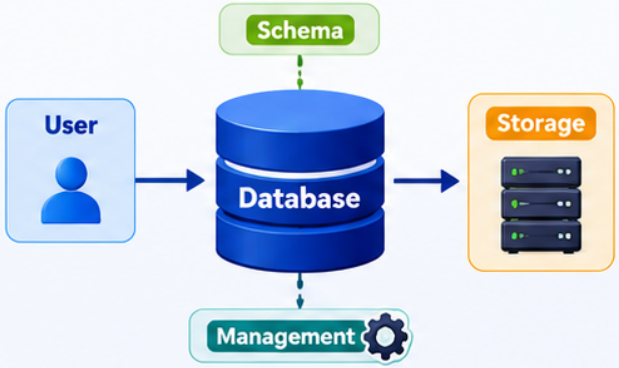
Syntax:

```
DROP DATABASE database_name;
```

Good Database Names

- Use lowercase letters
- Use underscores (_)
- Be short and meaningful
- Avoid spaces and special characters

How It Works



- Organized Data
- Better Performance
- Easy Management

Example

```
mysql> CREATE DATABASE company;
Query OK, 1 row affected (0.01 sec)

mysql> USE company;
Database changed

mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| company            |
| information_schema |
| mysql              |
+--------------------+
```

Quick Commands

Command	Example	Description
CREATE DATABASE	CREATE DATABASE shop;	Creates a new database
USE	USE shop;	Selects a database
SHOW DATABASES	SHOW DATABASES;	Shows all databases
DROP DATABASE	DROP DATABASE shop;	Deletes a database

Key Points

- A database is a container for tables and data.
- Use meaningful names for your databases.
- Always backup before dropping a database.

Benefits

- Better Organization
- Secure Management
- Improved Performance
- Easy Maintenance

★ "Create → Manage → Use → Secure | MySQL Makes Database Management Simple & Powerful!" ★

CREATE TABLE

Design Your Table Structure

1 CREATE TABLE SYNTAX

```
CREATE TABLE table_name (
    column1 data_type [constraint],
    column2 data_type [constraint],
    ...
    table_constraints
);
```

2 EXAMPLE

```
CREATE TABLE students (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  email VARCHAR(100) UNIQUE,
  age INT CHECK (age >= 0),
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

3 TABLE DESIGN FLOW

4 EXAMPLE TABLE STRUCTURE: *students*

Column Name	Data Type	Constraints	Description
id	INT	PRIMARY KEY, AUTO_INCREMENT	Unique ID for each student
name	VARCHAR(100)	NOT NULL	Student full name
email	VARCHAR(100)	UNIQUE	Student email address
age	INT	CHECK (age >= 0)	Student age
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Record creation time

5 COMMON DATA TYPES

INT	Integer numbers
VARCHAR(n)	Variable length string (Text)
CHAR(n)	Fixed length string
DATE	Date value (YYYY-MM-DD)
DATETIME	Date and time value
BOOLEAN	True or False (0 or 1)
DECIMAL(p,s)	Exact decimal numbers

6 CONSTRAINTS

PRIMARY KEY	Uniquely identifies each row.
NOT NULL	Column cannot have NULL values.
UNIQUE	All values in the column must be unique.
CHECK	Ensures values meet a condition.
DEFAULT	Sets a default value if none is provided.
FOREIGN KEY	Links to a primary key in another table.

7 PRIMARY KEY

- Uniquely identifies each row.
- Only one PRIMARY KEY per table.
- Cannot be NULL.

8 AUTO_INCREMENT

- Automatically generates a unique number.
- Usually used with PRIMARY KEY.

9 SCHEMA

- Schema defines the structure of the database.
- Includes tables, columns, relationships and rules.

10 TABLE DESIGN TIPS

- Keep it simple and meaningful.
- Use appropriate data types.
- Define keys and constraints.
- Plan for future scalability.

GOOD TABLE DESIGN = BETTER PERFORMANCE + DATA INTEGRITY

Well-structured tables make your database efficient, reliable and easy to maintain.

QUICK SUMMARY

Define Columns → Choose Data Types → Add Constraints → Create Table → Use & Manage

MySQL DATA TYPES

Choose the Right Data Type for Better Performance & Storage

DATA TYPE	DESCRIPTION	SYNTAX / EXAMPLE	SAMPLE VALUE	STORAGE	USE CASE
INT	Integer numbers (whole numbers)	`INT` `INT(11)`	`25`	4 bytes	ID, Count, Age, Quantity
VARCHAR	Variable length string (max length specified)	`VARCHAR(n)` `VARCHAR(100)`	`'John Doe'`	1 – 65,535 bytes	Names, Email, Addresses
CHAR	Fixed length string (max 255 characters)	`CHAR(n)` `CHAR(50)`	`'ABC '`	n bytes	Codes, Country Codes, Status
TEXT	Variable length string (large text data)	`TEXT`	`'This is a long text...'`	Up to 65,535 bytes	Descriptions, Comments
DATE	Date value (YYYY-MM-DD)	`DATE`	`'2025-06-16'`	3 bytes	Birth Date, Order Date
DATETIME	Date and time value (YYYY-MM-DD HH:MM:SS)	`DATETIME`	`'2025-06-16 14:30:00'`	8 bytes	Created At, Updated At
BOOLEAN	Boolean value (True or False)	`BOOLEAN` `TINYINT(1)`	`1 (True)`	1 byte	Status, Active, Is Verified
DECIMAL	Exact decimal numbers (fixed precision)	`DECIMAL(m, d)` `DECIMAL(10,2)`	`1234.56`	Varies (m + 2 bytes)	Prices, Amounts, Financial Data

QUICK TIPS

- Use INT for whole numbers.
- Use VARCHAR for variable length text.
- Use CHAR for fixed length codes.
- Use DATE for dates and DATETIME for date + time.
- Use BOOLEAN or TINYINT(1) for true/false.
- Use DECIMAL for accurate financial data.

STRING TYPES COMPARISON

TYPE	LENGTH	PADDING	BEST FOR
CHAR(n)	Fixed	Yes	Codes, Fixed Size Data
VARCHAR(n)	Variable	No	Names, Text, Email
TEXT	Variable (Large)	No	Long Text Data

NUMERIC TYPES COMPARISON

TYPE	EXAMPLE	DESCRIPTION
INT	123	Whole numbers (Integer)
DECIMAL(10,2)	1234.56	Exact decimal (Precision)
FLOAT	1234.567	Approximate decimal (Floating point)

CHOOSE SMARTLY
Right data type = Better Performance & Optimized Storage

PERFORMANCE
Optimized data types improve query speed and indexing.

STORAGE SAVING
Use appropriate types to save storage and reduce cost.

DATA INTEGRITY
Correct types ensure accurate and consistent data.

BEST PRACTICE
Plan your schema well for scalable and maintainable databases.

INSERT DATA

Add New Records to Your Table

1 INSERT INTO

Used to add new records into a table.

2 VALUES

Specifies the data values to be inserted.

3 MULTIPLE ROWS

Insert multiple records in one query.

4 NULL

Use NULL to insert missing or unknown values.

5 DEFAULT VALUE

If no value is provided, the default value is used.

SYNTAX

```
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);
```

EXAMPLES

1. Insert Single Row

```
INSERT INTO students (name, age, city)
VALUES ('Aman', 21, 'Amritsar');
```

2. Insert Multiple Rows

```
INSERT INTO students (name, age, city)
VALUES
 ('Priya', 20, 'Ludhiana'),
 ('Rahul', 22, 'Jalandhar'),
 ('Neha', 19, 'Patiala');
```

TABLE: students

id	name	age	city	course	email
1	Aman	21	Amritsar	BCA	aman@example.com
2	Priya	20	Ludhiana	B.Sc	priya@example.com
3	Rahul	22	Jalandhar	MCA	rahul@example.com
4	Neha	19	Patiala	B.Com	neha@example.com

DATA ENTRY → RECORDS ADDED

4 NULL EXAMPLE

```
INSERT INTO students (name, age, email)
VALUES ('Unknown', NULL, NULL);
```

NULL is stored when no value is provided.

5 DEFAULT VALUE EXAMPLE

```
-- email has DEFAULT 'not_provided@demo.com'
INSERT INTO students (name, age, city)
VALUES ('Arjun', 23, 'Bathinda');
```

If email is not specified, default value is used.

QUICK TIPS

- Always specify column names in INSERT.
- The order of values must match the columns.
- Use NULL for unknown or missing data.
- Use DEFAULT to set automatic values.
- You can insert multiple rows for better performance.

KEY POINTS

TABLE
Data is stored in tables (rows & columns).

INSERT
INSERT adds new records to the table.

MULTIPLE ROWS
Insert many records in one single query.

ACCURACY
Ensure correct data entry to avoid errors.

DATA INTEGRITY
Proper data entry maintains your database quality.

PERFORMANCE
Batch inserts improve performance.

SELECT DATA

Retrieve Data from Your Tables

1 SELECT

SELECT is used to retrieve data from a table.

2 FROM

FROM specifies the table to retrieve data from.

3 * (ALL COLUMNS)

* selects all columns from the table.

4 COLUMN NAME

Specify column name(s) to retrieve specific data.

5 ALIAS

Alias gives a temporary name to a column or table.

6 ROWS & RESULT

The query returns rows as a result set.

SYNTAX

```
SELECT column1, column2, ...
FROM   table_name
WHERE  condition;
```

- SELECT - columns to retrieve
- FROM - table name
- WHERE - filter conditions (optional)

EXAMPLES

1. Select All Columns

```
SELECT *
FROM students;
```

Retrieves all columns and rows.

2. Select Specific Columns

```
SELECT name, age, city
FROM students;
```

Retrieves only name, age and city columns.

3. With Alias

```
SELECT name AS student_name,
       age AS student_age
FROM students;
```

Retrieves columns with temporary names (alias).

4. With Condition

```
SELECT *
FROM students
WHERE age > 20;
```

Retrieves rows where age is greater than 20.

EXAMPLE RESULT

```
SELECT id, name, age, city
FROM students;
```

SQL QUERY → RESULT

id	name	age	city	course
1	Aman	21	Amritsar	BCA
2	Priya	20	Ludhiana	B.Sc
3	Rahul	22	Jalandhar	MCA
4	Neha	19	Patiala	B.Com

4 rows returned (result set)

SELECT COMPONENTS

Component	Description	Example
SELECT	Columns to retrieve	SELECT name, age
FROM	Table to retrieve data from	FROM students
*	All columns	SELECT * FROM students
COLUMN NAME	Specific column(s)	SELECT name, age
ALIAS	Temporary name	SELECT name AS n
ROWS	Each record in result	1, 2, 3, ...
RESULT	Output returned by query	Result set (table)

QUICK TIPS

- Use * only for quick testing; specify columns in real queries.
- Use meaningful alias names with AS.
- Use WHERE clause to filter unwanted rows.
- Use ORDER BY to sort results.
- Use LIMIT to restrict number of rows.

COMMON CLAUSES WITH SELECT

WHERE	- Filter rows based on condition
ORDER BY	- Sort the result set
GROUP BY	- Group rows with same values
HAVING	- Filter groups (used with GROUP BY)
LIMIT	- Limit number of rows returned

SAMPLE QUERIES

- SELECT * FROM students;
- SELECT name, city FROM students;
- SELECT name AS student_name, age FROM students;
- SELECT * FROM students WHERE city = 'Amritsar';
- SELECT name, age FROM students ORDER BY age DESC;
- SELECT * FROM students LIMIT 3;

KEY POINTS

- SELECT is the most basic and important SQL command.
- It helps to view, analyze and report data.
- Better SELECT queries = Better database performance.

BEST PRACTICES

- Always specify columns instead of * in production.
- Use meaningful aliases for clarity.
- Use proper filtering and sorting for accurate results.
- Test queries with LIMIT before full execution.

WHERE CLAUSE

Filter Data with Conditions

WHAT IS WHERE CLAUSE?

The WHERE clause is used to filter records and return only the rows that meet a specific condition.

BASIC SYNTAX

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

KEY BENEFITS

- Filter unwanted data
- Get precise results
- Improve performance
- Combine with other clauses
- Powerful data retrieval

COMPARISON OPERATORS

Operator	Meaning	Example	Description
=	Equal to	age = 25	Returns rows where values are equal
!= or <>	Not equal to	age != 25	Returns rows where values are not equal
>	Greater than	age > 25	Returns rows where values are greater
<	Less than	age < 25	Returns rows where values are less
>=	Greater than or equal to	age >= 25	Returns rows where values are greater than or equal
<=	Less than or equal to	age <= 25	Returns rows where values are less than or equal

WHERE CLAUSE EXAMPLES

1. Equal To

```
SELECT * FROM students
WHERE age = 21;
```

Result: Aman, Simran

2. Not Equal To

```
SELECT * FROM students
WHERE age != 20;
```

Result: Aman, Rahul, Neha, Arjun, Simran

3. Greater Than

```
SELECT * FROM students
WHERE marks > 80;
```

Result: Rahul, Arjun, Simran

4. Less Than

```
SELECT * FROM students
WHERE age < 21;
```

Result: Priya, Neha

5. Greater Than or Equal To

```
SELECT * FROM students
WHERE age >= 22;
```

Result: Rahul, Arjun

5. Between (Range)

```
SELECT * FROM students
WHERE age BETWEEN 20 AND 22;
```

Result: Aman, Priya, Rahul, Simran

EXAMPLE TABLE: *students*

id	name	age	city	course	marks
1	Aman	21	Amritsar	BCA	78
2	Priya	20	Ludhiana	B.Sc	68
3	Rahul	22	Jalandhar	MCA	85
4	Neha	19	Patiala	B.Com	72
5	Arjun	24	Amritsar	BCA	90
6	Simran	21	Ludhiana	MCA	88

USING LOGICAL OPERATORS

Operator	Meaning	Example	Description
AND	All conditions must be true	age > 20 AND marks > 70	Returns rows where both conditions are true
OR	Any one condition must be true	city = 'Amritsar' OR city = 'Ludhiana'	Returns rows where any one condition is true
NOT	Reverses the condition	NOT age > 20	Returns rows where condition is false

MORE EXAMPLES

- `SELECT * FROM students WHERE city = 'Amritsar';` — Result: Aman, Arjun
- `SELECT * FROM students WHERE course = 'MCA' AND marks > 80;` — Result: Rahul, Simran
- `SELECT * FROM students WHERE name LIKE 'A%';` — Result: Aman, Arjun
- `SELECT * FROM students WHERE marks BETWEEN 70 AND 85;` — Result: Aman, Rahul, Neha

QUICK TIPS

- Use WHERE to filter and get only required data.
- Combine multiple conditions using AND, OR, NOT.
- Use comparison operators correctly for accurate results.
- Use BETWEEN for range filtering.
- Indexes on columns used in WHERE improve performance.

KEY POINTS

- WHERE filters the data
- Use operators to set conditions
- Returns only matching rows
- Better queries = Better performance

ORDER BY & LIMIT

Sort and Control Your Results

1 ORDER BY

ORDER BY is used to sort the result set in ascending or descending order.

ASC (Ascending)

Sorts data in ascending order (A to Z, 1 to 9).

Default order.

DESC (Descending)

Sorts data in descending order (A, 9 to 1).

2 LIMIT

LIMIT is used to restrict the number of rows returned.

3 OFFSET

OFFSET is used with **LIMIT** to skip a certain number of rows.

SYNTAX

ORDER BY Syntax

```
SRLECT column1, column2, ...
FROM table_name
ORDER BY column_name [ASC|DESC];
```

LIMIT Syntax

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column_name [ASC|DESC]
LIMIT number_of_rows [OFFSET offset_value];
```

ORDER BY EXAMPLES

1. Ascending Order (ASC)

```
SELECT * FROM students
ORDER BY name ASC;
```

id	name	age	city	marks
1	Aman	21	Amritsar	78
5	Arjun	24	Amritsar	90
3	Rahul	22	Jalandhar	85
2	Priya	20	Ludhiana	68
6	Simran	21	Ludhiana	88
4	Neha	19	Patiala	72

2. Descending Order (DESC)

```
SELECT * FROM students
ORDER BY marks DESC;
```

id	name	age	city	marks
5	Arjun	24	Amritsar	90
6	Simran	21	Ludhiana	88
3	Rahul	22	Jalandhar	85
1	Aman	21	Amritsar	78
4	Neha	19	Patiala	72
2	Priya	20	Ludhiana	68

You can sort by multiple columns. `SELECT * FROM students ORDER BY city ASC, marks DESC;`

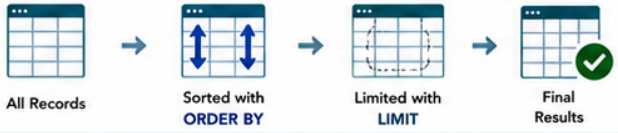
SAMPLE TABLE: *students*

id	name	age	city	marks
1	Aman	21	Amritsar	78
2	Priya	20	Ludhiana	68
3	Rahul	22	Jalandhar	85
4	Neha	19	Patiala	72
5	Arjun	24	Amritsar	90
6	Simran	21	Ludhiana	88

LIMIT & OFFSET EXAMPLES

1 First 3 Records	`SELECT * FROM students ORDER BY id ASC LIMIT 3;`	Result: 1, 2, 3
2 Next 3 Records (Pagination)	`SELECT * FROM students ORDER BY id ASC LIMIT 3 OFFSET 3;`	Result: 4, 5, 6
3 Top 2 by Marks	`SELECT * FROM students ORDER BY marks DESC LIMIT 2;`	Result: 5, 6
4 Skip 2, Take 2	`SELECT * FROM students ORDER BY marks DESC LIMIT 2 OFFSET 2;`	Result: 3, 1

HOW ORDER BY & LIMIT WORK TOGETHER



PAGINATION EXAMPLE (Page Size = 2)

QUICK TIPS

- Use **ORDER BY** to sort your results meaningfully.
- **ASC** is default (you can omit it).
- Use **LIMIT** to improve performance and control output.
- Use **LIMIT** with **OFFSET** for pagination.
- Always use **ORDER BY** with **LIMIT** for consistent results.

KEY POINTS

ORDER BY
Sorts the result set (ASC or DESC).

ASC / DESC
ASC = Ascending
DESC = Descending.

LIMIT
Restricts the number of rows returned.

OFFSET
Skips a number of rows before starting.

PAGINATION
Use LIMIT + OFFSET to navigate pages.

BETTER RESULTS
Sorting + Limiting = Fast, clean & useful data.

UPDATE DATA

Modify Existing Data in Your Table

1 UPDATE

UPDATE is used to modify existing records in a table.

2 SET

SET specifies the column(s) and the new value(s) to update.

3 WHERE

WHERE specifies which record(s) to update. (Omit WHERE with caution!)

MODIFY DATA

Changes the data in specified column(s) for the selected record(s).

SAVE CHANGES

After executing UPDATE, changes are saved in the database.

SYNTAX

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

IMPORTANT

Always use WHERE clause to avoid updating all rows!

EXAMPLES

1. Update Single Column

```
UPDATE students
SET city = 'Ludhiana'
WHERE id = 1;
```

Updates city for the student with id = 1.

2. Update Multiple Columns

```
UPDATE students
SET city = 'Amritsar',
    course = 'MCA'
WHERE id = 2;
```

Updates city and course for the student with id = 2.

3. Update with Condition

```
UPDATE students
SET marks = marks + 5
WHERE course = 'BCA';
```

Increases marks by 5 for all students in BCA course.

BEFORE & AFTER UPDATE

Before Update (id = 1)

id	name	age	city	course	marks
1	Aman	21	Amritsar	BCA	78

UPDATE Query Executed

After Update (id = 1)

id	name	age	city	course	marks
1	Aman	21	Ludhiana	BCA	78

KEY POINTS

- Always use WHERE clause to update specific rows.
- You can update one or multiple columns at a time.
- You can use functions and expressions in SET.
- Check your data before updating.
- Use SELECT first to preview the affected rows.

REAL-WORLD EXAMPLE: Change Course of Student

1. Check Data
```
SELECT * FROM students WHERE id = 3;
```
2. Update Data
```
UPDATE students SET course = 'BCA' WHERE id = 3;
```
3. Verify Update
```
SELECT * FROM students WHERE id = 3;
```
4. Changes Saved

Data is updated and saved permanently.

SAMPLE TABLE: *students*

id	name	age	city	course	marks
1	Aman	21	Amritsar	BCA	78
2	Priya	20	Ludhiana	B.Sc	68
3	Rahul	22	Jalandhar	MCA	85
4	Neha	19	Patiala	B.Com	72
5	Arjun	24	Amritsar	BCA	90

COMMON UPDATE SCENARIOS

Update One Column	`UPDATE students SET city = 'Patiala' WHERE id = 4;`
Update Multiple Columns	`UPDATE students SET city = 'Amritsar', marks = 95 WHERE id = 5;`
Update with Condition	`UPDATE students SET marks = marks + 10 WHERE course = 'MCA';`
Update Many Rows	`UPDATE students SET city = 'Ludhiana' WHERE city = 'Amritsar';`

CAUTION

```
UPDATE students SET city = 'Ludhiana';
```

This will update ALL rows in the table!
Always use WHERE clause.

QUICK REFERENCE

UPDATE
UPDATE table_name
(Modify existing data)

SET
SET column = value
(Specify new value)

WHERE
WHERE condition
(Select rows to update)

→ **SAVE**
Changes are saved in the database

BEST PRACTICES

- Always Back Up before updating data.
- Use WHERE to avoid accidental updates.
- Test on small data before running on production.
- Verify after update to ensure accuracy.

DELETE & TRUNCATE

Remove Data, Reset Tables & Keep Your Database Clean

1 DELETE

DELETE

Removes selected records from a table using WHERE condition.

2 TRUNCATE

TRUNCATE

Removes ALL records from a table.
Faster than DELETE.

3 DROP TABLE

DROP TABLE

Removes the entire table structure along with all data, indexes, etc.

4 KEY DIFFERENCES

Understand the difference before using.
Choose the right command!

5 CLEANUP & MAINTENANCE

Keep your database clean, optimized and running smoothly.

SYNTAX

DELETE Syntax

```
DELETE FROM table_name
WHERE condition;
```

TRUNCATE Syntax

```
TRUNCATE TABLE table_name;
```

DROP TABLE Syntax

```
DROP TABLE table_name;
```

Always double-check before executing these commands. Some actions cannot be undone!

EXAMPLES

1. DELETE Specific Records

```
DELETE FROM students
WHERE age < 20;
```

Removes only students with age less than 20.

2. TRUNCATE Table

```
TRUNCATE TABLE students;
```

Removes ALL records from students table. Table structure remains.

3. DROP Table

```
DROP TABLE students;
```

Deletes the entire students table permanently. Data + Structure + Indexes

DELETE vs TRUNCATE vs DROP TABLE

Feature	DELETE	TRUNCATE	DROP TABLE
Removes	Selected Records	All Records	Table + All Data
WHERE Clause	Yes	No	No
Table Structure	Remains	Remains	Removed
Indexes	Remain	Remain	Removed
AUTO_INCREMENT	Not Reset	Reset	Removed
Triggers	Fire (per row)	Do Not Fire	Removed
Rollback (Transaction)	Yes	No (Auto Commit)	No (Auto Commit)
Speed	Slower	Very Fast	Fast
Use Case	Remove some rows	Remove all rows	Delete table completely

SAMPLE TABLE: *students*

id	name	age	city	course
1	Aman	21	Amritsar	BCA
2	Priya	20	Ludhiana	B.Sc
3	Rahul	22	Jalandhar	MCA
4	Neha	19	Patiala	B.Com
5	Arjun	24	Amritsar	BCA

WHAT HAPPENS?

AFTER DELETE (age < 20)

id	name	age	city
1	Aman	21	Amritsar
2	Priya	20	Ludhiana
3	Rahul	22	Jalandhar
5	Arjun	24	Amritsar

Selected records removed.

AFTER TRUNCATE

id	name	age	city	course

All records removed.
Table structure remains.

AFTER DROP TABLE

Table 'students' does not exist.

Table + Data + Indexes permanently deleted.

QUICK COMMANDS

```
DELETE FROM students WHERE age < 20;
TRUNCATE TABLE students;
DROP TABLE students;
```

IMPORTANT TIPS

- Use DELETE when you need to remove specific records.
- Use TRUNCATE to quickly remove all records and reset AUTO_INCREMENT.
- Use DROP TABLE only when the table is no longer needed.
- Always backup your database before performing these operations!

CAUTION

- TRUNCATE and DROP TABLE cannot be rolled back.
- Data removed by DELETE (without WHERE) will delete ALL rows.
- Always test your queries with SELECT first.
- There is NO UNDO for DROP TABLE.

WHEN TO USE?

DELETE	When you want to remove some specific records.
TRUNCATE	When you want to remove all records, keep structure.
DROP TABLE	When you no longer need the table at all.

KEY POINTS

- **DELETE** Removes selected records with WHERE.
- **TRUNCATE** Removes all records, resets table.
- **DROP TABLE** Deletes table and everything in it.
- TRUNCATE is faster than DELETE.
- Structure remains in TRUNCATE & DELETE, but not in DROP TABLE.
- Always be careful, there is no UNDO for TRUNCATE or DROP TABLE.

CONSTRAINTS

Control Data Integrity & Accuracy

PRIMARY KEY
Uniquely identifies each record in a table.
No NULL values allowed.

FOREIGN KEY
Creates a relationship between two tables. Value must exist in the referenced table.

UNIQUE
Ensures all values in a column or combination of columns are unique.

NOT NULL
Ensures a column cannot have NULL values.

DEFAULT
Sets a default value for a column if no value is provided.

CHECK
Ensures that all values in a column satisfy a specific condition.

AUTO_INCREMENT
Automatically generates a unique number for new records (typically for PK).

- Always use **PRIMARY KEY** for each table.
- Use **FOREIGN KEY** to maintain relationships.
- Use **NOT NULL** for important data.
- Use **CHECK** to validate ranges and rules.
- Use **DEFAULT** to simplify data entry.
- Test your constraints with valid & invalid data.

SYNTAX OVERVIEW

```
CREATE TABLE table_name (
  column1 datatype CONSTRAINT constraint_name constraint_type,
  column2 datatype CONSTRAINT constraint_name constraint_type,
  ...
);
```

Constraints help to:
- Enforce rules
- Prevent invalid data
- Maintain data integrity

EXAMPLE: *students* TABLE WITH CONSTRAINTS

```
CREATE TABLE students (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(50) NOT NULL,
  email VARCHAR(100) UNIQUE,
  age INT CHECK (age >= 5 AND age <= 100),
  city VARCHAR(50) DEFAULT 'Amritsar'
);
```

Column	Constraint
id	PRIMARY KEY, AUTO_INCREMENT
name	NOT NULL
email	UNIQUE
age	CHECK (5 to 100)
city	DEFAULT 'Amritsar'

CONSTRAINTS IN ACTION (INSERT EXAMPLES)

Constraint	Valid Example ✅	Invalid Example ❌	Error / Result
PRIMARY KEY	id = 1	id = 1 (duplicate)	Duplicate entry for primary key
FOREIGN KEY	dept_id = 1 (exists)	dept_id = 99 (not exists)	Cannot add or update a child row
UNIQUE	email = 'a@example.com'	email = 'a@example.com' (duplicate)	Duplicate entry for unique key
NOT NULL	name = 'Aman'	name = NULL	Column cannot be NULL
DEFAULT	city not provided	city = 'Amritsar' (default set)	Default value applied
CHECK	age = 25	age = 150	Check constraint failed
AUTO_INCREMENT	id not provided	id = NULL (manual)	Auto-generated value used

CONSTRAINTS SUMMARY TABLE

Constraint	Ensures	Allows NULL?	Unique?	Auto Value?	Relationship?
PRIMARY KEY	Unique row identification	No	Yes	Yes (with AI)	No
FOREIGN KEY	Referential integrity	Yes	No	No	Yes
UNIQUE	Unique values	Yes	Yes	No	No
NOT NULL	Column has a value	No	No	No	No
DEFAULT	Default value if none given	Yes	No	No	No
CHECK	Values follow a rule	Yes	No	No	No
AUTO_INCREMENT	Auto-generate numbers	No	Yes	Yes	No

PARENT & CHILD TABLE EXAMPLE (FOREIGN KEY)

departments (parent table)

id (PK)	dept_name
1	CSE
2	IT
3	ECE

employees (child table)

id (PK)	name	dept_id (FK)
1	Aman	1
2	Priya	1
3	Rahul	2
4	Neha	3

```
CREATE TABLE employees (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(50) NOT NULL,
  dept_id INT,
  CONSTRAINT fk_dept FOREIGN KEY (dept_id)
    REFERENCES departments(id)
    ON DELETE RESTRICT
);
```

dept_id in employees must exist in departments.id

QUICK SUMMARY

	Constraint	Purpose
	PRIMARY KEY	Unique ID for each record
	FOREIGN KEY	Enforce relationship between tables
	UNIQUE	All values must be unique
	NOT NULL	Column cannot be empty
	DEFAULT	Set default value if none provided
	CHECK	Value must satisfy a condition
123	AUTO_INCREMENT	Auto-generate unique numbers

KEY TAKEAWAYS

- Constraints keep your database accurate & reliable.
- They prevent invalid or inconsistent data.
- Better integrity means better performance.
- Design with constraints, worry less later!

ALTER TABLE

Update Table Structure & Schema Easily

ALTER TABLE OPERATIONS

1 ADD COLUMN
Add a new column to an existing table.

2 DROP COLUMN
Remove an existing column from a table.

3 MODIFY COLUMN
Modify the data type or size of a column.

4 CHANGE COLUMN
Change the name and data type of a column.

5 RENAME TABLE
Rename an existing table.

6 CONSTRAINTS
Add, drop or modify constraints (PK, FK, UNIQUE, etc.).

KEY BENEFITS

- Update table structure without losing data
- Flexible schema management
- Supports application growth
- Maintain data integrity

ALTER TABLE SYNTAX

```
ALTER TABLE table_name
operation [column_name] definition;
```

Common Operations

- ADD COLUMN
- DROP COLUMN
- MODIFY COLUMN
- CHANGE COLUMN
- RENAME TABLE
- CONSTRAINTS

SAMPLE TABLE: *students* (Before Alter)

id (PK)	name	email	age	city
1	Aman	aman@ex.com	21	Amritsar
2	Priya	priya@ex.com	20	Ludhiana
3	Rahul	rahul@ex.com	22	Jalandhar

EXAMPLES WITH SYNTAX & RESULT

1. ADD COLUMN

```
ALTER TABLE students
ADD COLUMN phone VARCHAR(15);
```

Result

id	name	email	age	city	phone
1	Aman	aman@ex.com	21	Amritsar	NULL

2. DROP COLUMN

```
ALTER TABLE students
DROP COLUMN city;
```

Result

id	name	email	age
1	Aman	aman@ex.com	21

3. MODIFY COLUMN

```
ALTER TABLE students
MODIFY COLUMN email VARCHAR(100);
```

Result

id	name	email	age	city
1	Aman	aman@ex.com	21	Amritsar

4. CHANGE COLUMN

```
ALTER TABLE students
CHANGE COLUMN age student_age INT;
```

Result

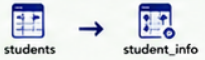
id	name	email	student_age	city
1	Aman	aman@ex.com	21	Amritsar

5. RENAME TABLE

```
ALTER TABLE students
RENAME TO student_info;
```

Result

Table name changed to: student_info



6. CONSTRAINTS EXAMPLE

```
ALTER TABLE students
ADD CONSTRAINT uq_email UNIQUE (email);
```

Result

Unique constraint added on email column.

UPDATE SCHEMA VISUALLY

Before		After (Example)	
id (PK)	INT	id (PK)	INT
name	VARCHAR(50)	name	VARCHAR(50)
email	VARCHAR(50)	email	VARCHAR(100)
age	INT	student_age	INT
city	VARCHAR(50)	phone	VARCHAR(15)

Modified · Added · Dropped

IMPORTANT NOTES

- ALTER TABLE does not remove data (except DROP COLUMN/RENAME).
- Always backup your data before making schema changes.
- Some changes may lock the table depending on storage engine.
- Use appropriate data types and constraints for data integrity.

QUICK REFERENCE

ADD COLUMN	`ALTER TABLE tbl ADD COLUMN col datatype;`
DROP COLUMN	`ALTER TABLE tbl DROP COLUMN col;`
MODIFY COLUMN	`ALTER TABLE tbl MODIFY COLUMN col datatype;`
CHANGE COLUMN	`ALTER TABLE tbl CHANGE old_col new_col datatype;`
RENAME TABLE	`ALTER TABLE tbl RENAME TO new_tbl;`
ADD CONSTRAINT	`ALTER TABLE tbl ADD CONSTRAINT name type (col);`
DROP CONSTRAINT	`ALTER TABLE tbl DROP CONSTRAINT name;`

CONSTRAINTS YOU CAN ADD

PRIMARY KEY	Ensures unique identity for each record
FOREIGN KEY	Maintains relationship between tables
UNIQUE	Ensures all values are unique
NOT NULL	Prevents NULL values in a column
DEFAULT	Sets a default value for the column
CHECK	Ensures values satisfy a condition

BEST PRACTICES

- Always take a backup before making changes.
- Test changes in a development environment first.
- Use constraints to maintain data accuracy.
- Plan schema changes during low traffic time.
- Keep your schema simple and consistent.

INDEXES

Speed Up Searches, Improve Performance

WHAT IS AN INDEX?

An INDEX is a special lookup table that the database engine uses to speed up data retrieval operations. Indexes reduce the number of rows MySQL has to scan.

BENEFITS OF INDEXES

- Faster search and data retrieval
- Improved query performance
- Efficient sorting and grouping
- Speeds up JOIN operations
- Better performance for large tables

WHEN TO USE INDEXES

Columns used in WHERE clause
Columns used in JOIN conditions
Columns used in ORDER BY
Columns with DISTINCT
Frequently searched columns

WHEN NOT TO USE INDEXES

- Very small tables
- Columns with low selectivity
- Columns with frequent updates (indexes slow down INSERT/UPDATE/DELETE)

TIP
Don't over-index. Too many indexes can slow down write operations and take more disk space.

INDEX SYNTAX OVERVIEW

```
CREATE INDEX index_name ON table_name (column_name);
CREATE UNIQUE INDEX index_name ON table_name (column_name);
DROP INDEX index_name ON table_name;
```

INDEX TIPS

- Use meaningful index names
- Index the right columns
- Avoid indexing too many columns
- Monitor and remove unused indexes

EXAMPLES

1. CREATE INDEX
Creates a basic index.

```
CREATE INDEX idx_name
ON students (name);
```

Result

id	name	age	city
1	Aman	21	Amritsar
2	Priya	20	Ludhiana
3	Rahul	22	Jalandhar
...	...	...	...

TABLE → INDEX (name)

Speeds up search on 'name' column

2. CREATE UNIQUE INDEX
Ensures all values are unique.

```
CREATE UNIQUE INDEX idx_email
ON students (email);
```

Result

id	email
1	aman@ex.com
2	priya@ex.com
3	rahul@ex.com
...	...

TABLE → UNIQUE INDEX (email)

Prevents duplicate email entries

3. DROP INDEX
Removes an existing index.

```
DROP INDEX idx_name
ON students;
```

Result

Index removed.

Index no longer used by the database.

4. MULTIPLE COLUMN INDEX
Index on more than one column.

```
CREATE INDEX idx_age_city
ON students (age, city);
```

Result

INDEX (age, city)

Useful for queries on (age, city) together

SAMPLE TABLE: *students*

id (PK)	name	email	age	city
1	Aman	aman@ex.com	21	Amritsar
2	Priya	priya@ex.com	20	Ludhiana
3	Rahul	rahul@ex.com	22	Jalandhar
4	Neha	neha@ex.com	19	Patiala
5	Arjun	arjun@ex.com	24	Amritsar

HOW INDEXES WORK (FAST SEARCH)

WITHOUT INDEX
(Full Table Scan)

Row 1
Row 2
Row 3
Row 4
Row 5
...
Row N

Database checks every row one by one.

WITH INDEX
(Indexed Lookup)

Database uses index to jump directly to the required rows.

PERFORMANCE COMPARISON

Operation	Without Index	With Index
SELECT with WHERE	Slow (Full table scan)	Fast (Indexed lookup)
JOIN operations	Slow	Fast
ORDER BY	Slow	Fast
INSERT / UPDATE / DELETE	Faster	Slower (index maintenance)
Storage Space	Less	More (extra space for indexes)

TYPES OF INDEXES

INDEX	Standard index. Allows duplicate values.
UNIQUE INDEX	Ensures all values are unique.
PRIMARY KEY	Automatically creates a unique index.
FULLTEXT INDEX	Used for full-text search (TEXT, VARCHAR).
COMPOSITE INDEX	Index on multiple columns.

CHECK EXISTING INDEXES

```
SHOW INDEX FROM students;
```

Result

Table	Non_unique	Key_name	Column_name	Index_type
students	0	PRIMARY	id	BTREE
students	1	idx_name	name	BTREE
students	0	idx_email	email	BTREE
students	1	idx_age_city	age, city	BTREE

CREATE INDEX
`CREATE INDEX idx_name ON table (column);`
Create a basic index

CREATE UNIQUE INDEX
`CREATE UNIQUE INDEX idx_name ON table (column);`
Create a unique index

DROP INDEX
`DROP INDEX idx_name ON table;`
Remove an index

Better Performance
Faster queries and retrieval

Fast Search
Quick lookup with less time

Optimization
Improves overall database efficiency

Scalability
Handles large data with ease

Smart Choice
Use indexes wisely!

AGGREGATE FUNCTIONS

Powerful Functions for Statistics, Reports & Data Analysis

WHAT ARE AGGREGATE FUNCTIONS?

Aggregate functions perform a calculation on a set of values and return a single result.
They are essential for statistics, reports and data analysis.

SYNTAX OVERVIEW

```
SELECT   AGGREGATE_FUNCTION(column_name)
FROM     table_name
WHERE    condition
GROUP BY column_name
HAVING   condition;
```

BENEFITS

- Quick summary of large data
- Helps in decision making
- Improves performance
- Essential for reports & dashboards

SAMPLE TABLE: sales

id	product	category	quantity	price
1	Laptop	Electronics	2	55000
2	Mobile	Electronics	5	15000
3	Keyboard	Accessories	10	800
4	Mouse	Accessories	15	500
5	Monitor	Electronics	3	12000
6	Speaker	Accessories	7	2000

COMMON AGGREGATE FUNCTIONS

COUNT()	Returns the number of rows.
SUM()	Returns the total sum of numeric values.
AVG()	Returns the average of numeric values.
MIN()	Returns the smallest value.
MAX()	Returns the largest value.

KEY POINTS

- Aggregate functions ignore NULL values.
- Use GROUP BY to get results per group.
- Use HAVING to filter groups.
- Use alias for better readability.

AGGREGATE FUNCTIONS EXAMPLES

1. COUNT() – Total Rows

```
SELECT COUNT(*) AS total_rows
FROM sales;
```

Result

total_rows
6

2. SUM() – Total Quantity

```
SELECT SUM(quantity) AS total_qty
FROM sales;
```

Result

total_qty
42

3. AVG() – Average Price

```
SELECT AVG(price) AS avg_price
FROM sales;
```

Result

avg_price
10050.00

4. MIN() – Minimum Price

```
SELECT MIN(price) AS min_price
FROM sales;
```

Result

min_price
500

5. MAX() – Maximum Price

```
SELECT MAX(price) AS max_price
FROM sales;
```

Result

max_price
55000

GROUP BY EXAMPLES (Statistics by Category)

Get total quantity and average price by category

```
SELECT category,
       COUNT(*) AS total_items,
       SUM(quantity) AS total_qty,
       AVG(price) AS avg_price,
       MIN(price) AS min_price,
       MAX(price) AS max_price
FROM sales
GROUP BY category;
```

Result

category	total_items	total_qty	avg_price	min_price	max_price
Electronics	3	10	27333.33	12000	55000
Accessories	3	32	1100.00	500	2000

HAVING EXAMPLE (Filter Groups)

Show categories having total quantity > 20

```
SELECT category, SUM(quantity) AS total_qty
FROM sales
GROUP BY category
HAVING SUM(quantity) > 20;
```

Result

category	total_qty
Accessories	32

REAL-WORLD USE CASES

Business Reports
Total sales, average order value, product performance.

Analytics Dashboards
KPIs, trends, summaries and growth metrics.

Inventory Analysis
Total stock, fast moving items, low stock alerts.

AGGREGATE FUNCTIONS WITH NULL VALUES

NULL values are ignored by aggregate functions (except COUNT(*)).

Function	With NULL Values (example: 1, 2, NULL, 4)
COUNT(column)	3 (counts 1,2,4)
COUNT(*)	4 (counts all rows)
SUM(column)	7 (1 + 2 + 4)
AVG(column)	2.33 (7 / 3)
MIN(column)	1
MAX(column)	4

QUICK REFERENCE

Function	Description
COUNT()	Returns number of rows
SUM()	Returns total sum
AVG()	Returns average value
MIN()	Returns minimum value
MAX()	Returns maximum value

TIPS

- Use meaningful aliases for results.
- Combine with WHERE, GROUP BY, HAVING for powerful reports.
- Indexes on columns used in GROUP BY improve performance.
- Test queries on small data first.

EXAMPLE: Total Sales Amount

```
SELECT SUM(quantity * price)
       AS total_sales
FROM   sales;
```

Result

total_sales
31600

GROUP BY & HAVING

Organize Data, Filter Groups & Create Powerful Reports

DATA ANALYSIS REPORTS

WHAT IS GROUP BY?

GROUP BY is used to arrange identical data into groups.
It is often used with aggregate functions.

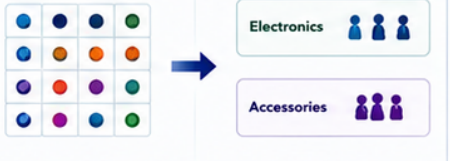
WHAT IS HAVING?

HAVING is used to filter the groups created by GROUP BY.
It works like WHERE but for groups.

COMMON AGGREGATE FUNCTIONS

COUNT()	Returns number of rows in each group.
SUM()	Returns total sum of values in each group.
AVG()	Returns average value in each group.
MIN()	Returns minimum value in each group.
MAX()	Returns maximum value in each group.

VISUAL: HOW GROUP BY WORKS



SYNTAX OVERVIEW

```
SELECT  column1, aggregate_function(column2)
FROM    table_name
WHERE   condition
GROUP BY column1, column2, ...
HAVING  aggregate_condition
ORDER BY column1;
```

EXECUTION ORDER

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
6. ORDER BY

SAMPLE TABLE: sales

id	product	category	quantity	price	sales_date
1	Laptop	Electronics	2	55000	2024-04-01
2	Mobile	Electronics	5	15000	2024-04-01
3	Keyboard	Accessories	10	800	2024-04-02
4	Mouse	Accessories	15	500	2024-04-02
5	Monitor	Electronics	3	12000	2024-04-03
6	Speaker	Accessories	7	2000	2024-04-03
7	Tablet	Electronics	4	18000	2024-04-03

EXAMPLES: GROUP BY & HAVING IN ACTION

1. GROUP BY Category (Total Quantity)

```
SELECT category, SUM(quantity) AS total_qty
FROM sales
GROUP BY category;
```

Result

category	total_qty
Electronics	14
Accessories	32

Groups data by category and calculates total quantity.

2. GROUP BY with COUNT() and AVG()

```
SELECT category,
       COUNT(*) AS total_items,
       AVG(price) AS avg_price
FROM sales
GROUP BY category;
```

Result

category	total_items	avg_price
Electronics	4	25000.00
Accessories	3	1100.00

Shows count of items and average price per category.

3. GROUP BY with HAVING (Filter Groups)

```
SELECT category, SUM(quantity) AS total_qty
FROM sales
GROUP BY category
HAVING SUM(quantity) > 20;
```

Result

category	total_qty
Accessories	32

Returns only those categories where total quantity is greater than 20.

KEY POINTS

- GROUP BY groups rows with same values in specified column(s).
- Aggregate functions are used with GROUP BY.
- HAVING filters the grouped records.
- WHERE filters individual rows before grouping.
- You can use multiple columns in GROUP BY.
- Always use meaningful aliases for better reports.

MORE EXAMPLES

#	Query	Description	Result (from sample table)
1	`SELECT category, MAX(price) AS max_price FROM sales GROUP BY category;`	Maximum price in each category	category: Electronics, 55000 / Accessories, 2000
2	`SELECT sales_date, COUNT(*) AS total_sales FROM sales GROUP BY sales_date;`	Count of sales per date	sales_date: 2024-04-01, 2 / 2024-04-02, 2 / 2024-04-03, 3
3	`SELECT category, AVG(quantity) AS avg_qty FROM sales GROUP BY category HAVING AVG(quantity) >= 5;`	Categories with average quantity >= 5	category: Accessories, avg_qty: 10.67

QUICK SUMMARY

Clause / Function	Purpose
GROUP BY	Groups rows with same values.
HAVING	Filters groups after aggregation.
COUNT()	Counts rows in each group.
SUM()	Calculates total sum in each group.
AVG()	Calculates average in each group.
MIN()	Finds minimum value in each group.
MAX()	Finds maximum value in each group.

PRO TIPS

- Use GROUP BY with indexes on large tables for better performance.
- Always test queries with small data first.
- Use HAVING instead of WHERE for conditions on aggregate functions.
- Great for reports, dashboards & business analysis.

LIKE & BETWEEN

Powerful Pattern Search & Filtering in MySQL

SEARCH FILTER REPORTS

WHAT IS LIKE?

LIKE is used to search for a specified pattern in a column.

```
SELECT * FROM table_name
WHERE column_name LIKE 'pattern';
```

WILDCARDS IN LIKE

Wildcard	Symbol	Meaning	Example	Matches
Any number of characters	%	Zero or more characters	'a%'	apple, a123, abc, a
Single character	_	Exactly one character	'a_e'	ace, age, ale

TIPS

- LIKE is case-insensitive by default in MySQL (case-sensitive depends on collation).
- Combine % and _ for advanced search.

SYNTAX OVERVIEW

```
-- LIKE
SELECT * FROM table_name
WHERE column_name LIKE 'pattern';

-- BETWEEN
SELECT * FROM table_name
WHERE column_name BETWEEN value1 AND value2;

-- IN
SELECT * FROM table_name
WHERE column_name IN (value1, value2, ...);

-- IS NULL
SELECT * FROM table_name
WHERE column_name IS NULL;

SELECT * FROM table_name
WHERE column_name IS NOT NULL;
```

OPERATORS OVERVIEW

%	Matches any number of characters
_	Matches exactly one character
BETWEEN	Matches a value within a range
IN	Matches any value in a list
IS NULL	Matches rows with NULL values

SAMPLE TABLE: users

id	name	email	age	city
1	Aman Singh	aman@gmail.com	21	Amritsar
2	Priya Sharma	priya@gmail.com	25	Ludhiana
3	Rahul Verma	rahul@gmail.com	28	Jalandhar
4	Neha Kaur	neha@gmail.com	22	Patiala
5	Arjun Mehta	arjun@gmail.com	30	Amritsar
6	Simranjeet Kaur	simran@gmail.com	27	Chandigarh
7	Vikram Roy	vikram@gmail.com	35	Delhi

LIKE - PATTERN SEARCH EXAMPLES

Query	Description	Result
`SELECT * FROM users WHERE name LIKE 'A%';`	Names starting with 'A'	Aman Singh, Arjun Mehta
`SELECT * FROM users WHERE name LIKE '%a';`	Names ending with 'a'	Priya Sharma, Neha Kaur
`SELECT * FROM users WHERE name LIKE '%an%';`	Names containing 'an'	Aman Singh
`SELECT * FROM users WHERE name LIKE '_eha%';`	Names with 2nd char 'eha'	Neha Kaur
`SELECT * FROM users WHERE name LIKE '____%';`	Names with at least 5 letters	All names
`SELECT * FROM users WHERE city LIKE 'A%';`	Cities starting with 'A'	Amritsar

BETWEEN - RANGE SEARCH EXAMPLES

Query	Description	Result
`SELECT * FROM users WHERE age BETWEEN 22 AND 30;`	Age between 22 and 30	Priya, Rahul, Neha, Arjun, Simranjeet
`SELECT * FROM users WHERE age BETWEEN 25 AND 35;`	Age between 25 and 35	Priya, Rahul, Arjun, Simranjeet, Vikram
`SELECT * FROM users WHERE id BETWEEN 1 AND 4;`	ID between 1 and 4	First 4 records
`SELECT * FROM users WHERE city BETWEEN 'A' AND 'L';`	City between 'A' and 'L'	Amritsar, Chandigarh, Jalandhar

IN - LIST MATCHING EXAMPLES

Query	Description	Result
`SELECT * FROM users WHERE city IN ('Amritsar','Patiala');`	City is Amritsar or Patiala	Aman, Neha, Arjun
`SELECT * FROM users WHERE age IN (21,25,30);`	Age is 21, 25 or 30	Aman, Priya, Arjun
`SELECT * FROM users WHERE id IN (1,3,5,7);`	ID in the given list	1, 3, 5, 7 records

IS NULL - NULL CHECK EXAMPLES

Query	Description	Result
`SELECT * FROM users WHERE email IS NULL;`	Rows where email is NULL	(Shows NULL emails)
`SELECT * FROM users WHERE email IS NOT NULL;`	Rows where email is NOT NULL	All rows (no NULL emails)

QUICK REFERENCE TABLE

Operator	Use	Example
LIKE	Pattern search	name LIKE 'A%'
%	Any number of characters	'a%'
_	Single character	'a_e'
BETWEEN	Range search	age BETWEEN 20 AND 30
IN	Match any value in list	city IN ('Amritsar','Delhi')
IS NULL	Check NULL values	email IS NULL

VISUAL: LIKE PATTERN EXAMPLES

Pattern	Visual	Matches
'A%'	A_____	Aman, Arjun
'%a'	_____a	Priya, Neha
'%an%'	__an_____	Aman
'_eha%'	_eha_____	Neha Kaur
'____%'	____ (5+ chars)	All names

REAL-WORLD USE CASES

- Search customers by name patterns.
- Filter data within a range (dates, prices, age).
- Get records from specific categories or lists.
- Find missing data using IS NULL.
- Create powerful reports & dashboards.

BEST PRACTICES

- Use LIKE with % at the end for better index usage.
- Avoid leading % (like '%abc') if possible – it slows down.
- Use BETWEEN for ranges (inclusive of both values).
- Use IN for multiple exact matches instead of OR conditions.
- Always handle NULLs carefully in your queries.

Better Filtering
Better Performance
Better Reports!

CASE & ALIASES

Write Smarter, Readable & Meaningful Queries

DATA QUERY REPORTS

WHAT IS CASE?

CASE is used to perform conditional logic in SQL queries. It works like IF...ELSE in programming.

CASE SYNTAX

```
CASE
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2
  WHEN condition3 THEN result3
  ELSE result_else
END
```

CASE FLOW

- Start
- Check condition 1 (WHEN)
- If TRUE → THEN result1
- Check next condition
- If TRUE → THEN resultN
- If all FALSE → ELSE result
- End (END)

CASE SYNTAX OVERVIEW

1. Simple CASE

```
SELECT column_name,
       CASE column_name
           WHEN value1 THEN result1
           WHEN value2 THEN result2
           ELSE result_else
       END AS alias_name
FROM table_name;
```

2. Searched CASE

```
SELECT column_name,
       CASE
           WHEN condition1 THEN result1
           WHEN condition2 THEN result2
           WHEN condition3 THEN result3
           ELSE result_else
       END AS alias_name
FROM table_name;
```

KEYWORDS

CASE	: Start CASE expression
WHEN	: Define condition
THEN	: Result when condition is TRUE
ELSE	: Result when all conditions FALSE
END	: End of CASE expression
AS	: Create an alias (new name)

SAMPLE TABLE: *employees*

id	name	department	salary
1	Aman	Sales	45000
2	Priya	HR	38000
3	Rahul	IT	60000
4	Neha	Sales	52000
5	Vikram	IT	75000
6	Simran	HR	42000

EXAMPLES WITH OUTPUT

1. Simple CASE Example

```
SELECT name, department,
       CASE department
           WHEN 'Sales' THEN 'Business Team'
           WHEN 'HR' THEN 'People Team'
           WHEN 'IT' THEN 'Tech Team'
           ELSE 'Other'
       END AS team_name
FROM employees;
```

name	department	team_name
Aman	Sales	Business Team
Priya	HR	People Team
Rahul	IT	Tech Team
Neha	Sales	Business Team
Vikram	IT	Tech Team
Simran	HR	People Team

2. Searched CASE Example

```
SELECT name, salary,
       CASE
           WHEN salary >= 60000 THEN 'High'
           WHEN salary >= 40000 THEN 'Medium'
           ELSE 'Low'
       END AS salary_level
FROM employees;
```

name	salary	salary_level
Aman	45000	Medium
Priya	38000	Low
Rahul	60000	High
Neha	52000	Medium
Vikram	75000	High
Simran	42000	Medium

3. Multiple Conditions

```
SELECT name, salary,
       CASE
           WHEN department = 'IT' AND salary > 65000
                THEN 'Top Performer'
           WHEN department = 'IT' THEN 'IT Staff'
           WHEN department = 'Sales' THEN 'Sales Staff'
           ELSE 'Other Staff'
       END AS performance
FROM employees;
```

name	salary	performance
Aman	45000	Sales Staff
Priya	38000	Other Staff
Rahul	60000	IT Staff
Neha	52000	Sales Staff
Vikram	75000	Top Performer
Simran	42000	Other Staff

ALIASES (AS)

Aliases are used to give a table or column a temporary name for better readability.

```
SELECT name AS employee_name,
       salary AS monthly_salary
FROM employees;
```

Without Alias		→	With Alias	
name	salary		employee_name	monthly_salary
Aman	45000		Aman	45000
Priya	38000		Priya	38000
...	...		...	...

WHY USE ALIASES?

- Makes queries easy to read and understand
- Useful in reports and dashboards
- Helps when using calculations or expressions
- Improves presentation of results

QUICK REFERENCE

Keyword	Purpose	Example
CASE	Start of CASE expression	CASE
WHEN	Add a condition	WHEN salary > 50000
THEN	Result if condition is TRUE	THEN 'High'
ELSE	Result if all conditions FALSE	ELSE 'Low'
END	End of CASE expression	END
AS	Create an alias	AS alias_name

BEST PRACTICES

- Use CASE for conditional logic in SELECT, ORDER BY, HAVING.
- Always use meaningful aliases with AS.
- Keep conditions clear and simple.
- Use ELSE to handle unmatched cases.

REAL-WORLD USE CASES

- Categorize salaries (High / Medium / Low)
- Group departments into teams
- Create custom labels in reports
- Show status (Active / Inactive)
- Add calculated columns with readable names

PRO TIP

You can use aliases with expressions and functions too!

```
SELECT name,
       salary * 12 AS annual_salary,
       ROUND(salary * 12, 2) AS annual_salary_rounded
FROM employees;
```

SQL VIEWS

Virtual Tables, Reusable Queries, Better Security & Powerful Reports

VIRTUAL TABLE · SECURITY · REPORTS · READ ONLY

WHAT IS A VIEW?

A View is a virtual table based on a result-set of an SQL statement.
It does not store data physically.
It shows data from one or more tables in a defined way.

KEY BENEFITS

- **Virtual Table** — Acts like a table but stores no data.
- **Reusable Query** — Save complex SELECT queries and reuse them.
- **Security** — Restrict access to sensitive data by exposing only required columns.
- **Simplify Reports** — Easier and cleaner reporting.
- **Read Only / Controlled Access** — Can be read-only or updatable (In some cases).

VIEW TYPES

- **Simple View** — Based on a single table.
- **Complex View** — Based on multiple tables, joins, conditions, groups, etc.

USE CASES

- Hide sensitive columns (e.g., salary, address)
- Simplify complex joins for reports
- Provide customized data to different users
- Create consistent reporting layer

SYNTAX OVERVIEW

CREATE VIEW Syntax

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Other View Commands

```
SHOW FULL TABLES WHERE Table_type = 'VIEW';
DESCRIBE view_name;
DROP VIEW view_name;
ALTER VIEW view_name AS SELECT ...;
```

EXAMPLE BASE TABLES

employees

id	name	department	salary	city
1	Aman	Sales	45000	Delhi
2	Priya	HR	38000	Mumbai
3	Rahul	IT	60000	Pune
4	Neha	Sales	52000	Delhi

departments

dept_id	dept_name	location
101	Sales	Delhi
102	HR	Mumbai
103	IT	Pune

EXAMPLE: CREATE & USE VIEW

1. Create a Simple View

```
CREATE VIEW sales_employees AS
SELECT id, name, salary, city
FROM employees
WHERE department = 'Sales';
```

Result (sales_employees)

id	name	salary	city
1	Aman	45000	Delhi
4	Neha	52000	Delhi

2. Create a Join View

```
CREATE VIEW emp_dept_view AS
SELECT e.id, e.name, d.dept_name, e.salary
FROM employees e
JOIN departments d ON e.department = d.dept_name;
```

Result (emp_dept_view)

id	name	dept_name	salary
1	Aman	Sales	45000
2	Priya	HR	38000
3	Rahul	IT	60000
4	Neha	Sales	52000

3. Use View Like a Table

```
SELECT * FROM sales_employees
WHERE salary > 46000;
```

Result

id	name	salary	city
4	Neha	52000	Delhi

```
SELECT dept_name, AVG(salary) AS avg_salary
FROM emp_dept_view
GROUP BY dept_name;
```

Result

dept_name	avg_salary
Sales	48500.00
HR	38000.00
IT	60000.00

MANAGING VIEWS

```
SHOW FULL TABLES WHERE Table_type = 'VIEW';
-- List all views

DESCRIBE view_name;
-- View structure

DROP VIEW view_name;
-- Delete a view

ALTER VIEW view_name AS SELECT ...;
-- Modify view
```

CAN WE UPDATE DATA THROUGH VIEW?

Possible	Not Possible
✓ Simple views (based on single table)	✗ Views with JOINs
✓ Without DISTINCT	✗ Views with GROUP BY
✓ Without GROUP BY	✗ Views with DISTINCT
✓ Without aggregation functions	✗ Views with Aggregates

SECURITY BENEFITS

Grant access only to views, not base tables.
Show only required columns/rows.
Protect sensitive data from direct access.

```
GRANT SELECT ON view_name TO 'user'@'host';
-- User can access only the view
```

VIEW VS TABLE

Feature	TABLE	VIEW
Stores Data	Yes	No (Virtual)
Disk Space	Consumes	No extra space
Performance	Faster (Direct)	Slightly slower
Security	Less control	Better control
Complex Queries	Harder	Easier (Predefined)

QUICK EXAMPLE SUMMARY

```
CREATE VIEW sales_employees AS
SELECT id, name, salary
FROM employees
WHERE department = 'Sales';

SELECT * FROM sales_employees;

DROP VIEW sales_employees;
```

PRO TIPS

- Use meaningful and descriptive names for views.
- Keep views simple and focused.
- Use views to standardize reports.
- Always test performance for complex views.
- Combine views with Stored Procedures for powerful solutions.

CREATE VIEW
`CREATE VIEW view_name AS SELECT ...;`
Create a virtual table

SELECT FROM VIEW
`SELECT * FROM view_name;`
Use view like a table

ALTER VIEW
`ALTER VIEW view_name AS SELECT ...;`
Modify the view

DROP VIEW
`DROP VIEW view_name;`
Remove the view

SHOW VIEWS
`SHOW FULL TABLES WHERE Table_type = 'VIEW';`
List all views

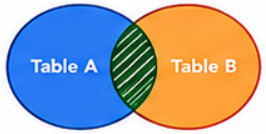
READ ONLY (Most Cases)
Views are mostly read-only for safety & consistency

INNER JOIN

Combine Data From Two Tables Based on Matching Records

TWO TABLES | MATCHING RECORDS | COMMON DATA RESULTS | RELATIONSHIP | QUERY RESULTS

WHAT IS INNER JOIN?



INNER JOIN returns only the rows that have matching values in both tables based on the join condition.

SYNTAX

```
SELECT column_list
FROM table1 AS t1
INNER JOIN table2 AS t2
ON t1.common_column = t2.common_column
WHERE condition;
```

SYNTAX BREAKDOWN

- `SELECT` : Columns to display
- `FROM` : First table
- `INNER JOIN` : Join with second table
- `ON` : Join condition
- `WHERE` : Filter results (optional)

SAMPLE TABLES

Table: employees

id	name	dept_id	salary
1	Aman	101	45000
2	Priya	102	38000
3	Rahul	103	60000
4	Neha	104	52000
5	Vikram	105	47000

Table: departments

dept_id	dept_name	location
101	Sales	Delhi
102	HR	Mumbai
103	IT	Pune
104	Finance	Delhi
106	Marketing	Bangalore

KEY POINTS

- Returns only matching records from both tables.
- Uses ON clause to define the relationship.
- Non-matching rows are excluded.
- Most commonly used JOIN in SQL.
- Improves data accuracy in results.

EXAMPLE QUERY & RESULT

Example Query (INNER JOIN)

```
SELECT e.id, e.name, d.dept_name, e.salary
FROM employees AS e
INNER JOIN departments AS d
 ON e.dept_id = d.dept_id
ORDER BY e.id;
```

Result (Matching Records Only)

id	name	dept_name	salary
1	Aman	Sales	45000
2	Priya	HR	38000
3	Rahul	IT	60000
4	Neha	Finance	52000

Only matching dept_id values from both tables are returned.

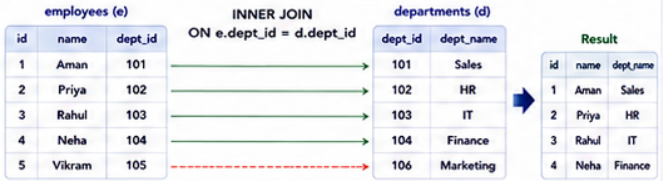
HOW INNER JOIN WORKS

Matches rows where `e.dept_id = d.dept_id` and returns the common data.

JOIN KEYWORD REFERENCE

Keyword	Purpose
INNER JOIN	Returns matching rows from both tables
ON	Specifies the join condition.
=	Used to match columns
AS	Gives alias (short name) to tables/columns

VISUAL REPRESENTATION



PRACTICAL EXAMPLE

Find all employees with their department name.

```
SELECT e.name AS employee_name,
       d.dept_name AS department,
       e.salary
FROM employees e
INNER JOIN departments d
ON e.dept_id  = d.dept_id;
```

Returns only employees who belong to a valid department.

COMMON USE CASES

- Get employee details with department information.
- Fetch order details with customer information.
- Show product details with category names.
- Generate reports from related data.

PRO TIP

Always ensure both tables have proper indexes on join columns for better performance.

QUICK SUMMARY

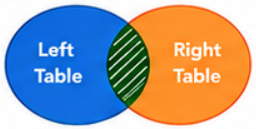
Join Type	Tables Involved	Match Condition	Returned Rows	Non-Matching Rows	Best For	Example Use	Result
INNER JOIN	Two Tables	ON clause	Matching Records Only	Excluded	Common/Related Data	Reports, Analysis, Dashboards	Clean, Relevant Data

LEFT JOIN

Return All Rows from Left Table and Matching Rows from Right Table

TWO TABLES · RELATIONSHIP · FILTER / ON · QUERY · RESULTS

WHAT IS LEFT JOIN?



LEFT JOIN returns all rows from the LEFT (table1) and the matching rows from the RIGHT table (table2). If no match is found, NULL values are returned for the right table.

SYNTAX

```
SELECT column_list
FROM left_table AS l
LEFT JOIN right_table AS r
ON l.common_column = r.common_column
WHERE condition;
```

SYNTAX BREAKDOWN

- SELECT : Columns to display
- FROM : First (left) table
- LEFT JOIN : Join with second (right) table
- ON : Join condition
- WHERE : Filter results (optional)

SAMPLE TABLES

Table: employees (LEFT TABLE)

id	name	dept_id	salary
1	Aman	101	45000
2	Priya	102	38000
3	Rahul	103	60000
4	Neha	104	52000
5	Vikram	105	47000

Table: departments (RIGHT TABLE)

dept_id	dept_name	location
101	Sales	Delhi
102	HR	Mumbai
103	IT	Pune
106	Marketing	Bangalore

KEY POINTS

- Returns all rows from the LEFT table.
- Returns matching rows from the RIGHT table.
- If no match, NULL values are returned for RIGHT table columns.
- Uses ON clause to define the relationship.
- Very useful when LEFT table data is mandatory.

EXAMPLE QUERY & RESULT

Example Query (LEFT JOIN)

```
SELECT e.id, e.name, d.dept_name, d.location
FROM employees AS e
LEFT JOIN departments AS d
ON e.dept_id = d.dept_id
ORDER BY e.id;
```

Result (All rows from LEFT + Matching RIGHT)

id	name	dept_id	dept_name	location
1	Aman	101	Sales	Delhi
2	Priya	102	HR	Mumbai
3	Rahul	103	IT	Pune
4	Neha	104	NULL	NULL
5	Vikram	105	NULL	NULL

Neha and Vikram have no matching department, so NULL values are returned for RIGHT table columns.

HOW LEFT JOIN WORKS

Take all rows from LEFT table.
Add matching rows from RIGHT table.
If no match, fill RIGHT table columns with NULL.

JOIN KEYWORD REFERENCE

Keyword	Purpose
LEFT JOIN	Returns all rows from LEFT table with matches from RIGHT
ON	Specifies the join condition
=	Used to match columns
AS	Gives alias (short name) to tables/columns

VISUAL REPRESENTATION

Employees (Left Table)

id	name	dept_id
1	Aman	101
2	Priya	102
3	Rahul	103
4	Neha	104
5	Vikram	105

LEFT JOIN
ON e.dept_id = d.dept_id

Departments (Right Table)

dept_id	dept_name
101	Sales
102	HR
103	IT
106	Marketing

⟷ = Match Found ⇢⊗ = No Match (NULL)

PRACTICAL EXAMPLE

Find all employees with their department details.

```
SELECT e.name AS employee_name,
       d.dept_name AS department_name
FROM employees e
LEFT JOIN departments d
ON e.dept_id = d.dept_id;
```

Useful for reports where LEFT table data should not be missed.

COMMON USE CASES

- Show all customers with their orders (if no order, still show customer).
- Display all employees with department info (even if department is not assigned).
- Generate master data reports with related details.

PRO TIP

Place the table you want ALL records from on the LEFT side of the JOIN.

QUICK SUMMARY

Join Type	Tables Involved	Returns	No Match	Join Condition	Relationship	Best For	Result
LEFT JOIN	Two Tables	All LEFT rows + Matching RIGHT rows	NULL values for RIGHT table	Defined using ON clause	Based on related columns	Reports, Listing, Master Data	Complete LEFT data with related info

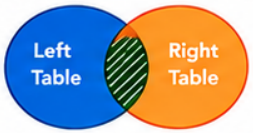
RIGHT JOIN

MySQL

TWO TABLES · RELATIONSHIP · FILTER / ON · QUERY · RESULTS

Return All Rows from Right Table and Matching Rows from Left Table

WHAT IS RIGHT JOIN?



RIGHT JOIN returns all rows from the RIGHT table and the matching rows from the LEFT table. If no match is found, NULL values are returned for the LEFT table columns.

SYNTAX

```
SELECT column_list
FROM left_table AS l
RIGHT JOIN right_table AS r
ON l.common_column = r.common_column
WHERE condition;
```

SYNTAX BREAKDOWN

- `SELECT` : Columns to display
- `FROM` : First (left) table
- `RIGHT JOIN` : Join with second (right) table
- `ON` : Join condition
- `WHERE` : Filter results (optional)

SAMPLE TABLES

Table: employees (LEFT TABLE)

id	name	dept_id	salary
1	Aman	101	45000
2	Priya	102	38000
3	Rahul	103	60000
4	Neha	104	52000
5	Vikram	105	47000

Table: departments (RIGHT TABLE)

dept_id	dept_name	location
101	Sales	Delhi
102	HR	Mumbai
103	IT	Pune
106	Marketing	Bangalore

KEY POINTS

- Returns all rows from the RIGHT table.
- Returns matching rows from the LEFT table.
- If no match, NULL values are returned for LEFT table columns.
- Uses ON clause to define the relationship.
- Very useful when RIGHT table data is mandatory.

EXAMPLE QUERY & RESULT

Example Query (RIGHT JOIN)

```
SELECT l.id, l.name, l.dept_id, d.dept_name, d.location
FROM employees AS l
RIGHT JOIN departments AS d
ON l.dept_id = d.dept_id
ORDER BY d.dept_id;
```

Result (All rows from RIGHT + Matching LEFT)

id	name	dept_id	dept_name	location
1	Aman	101	Sales	Delhi
2	Priya	102	HR	Mumbai
3	Rahul	103	IT	Pune
NULL	NULL	106	Marketing	Bangalore

Department 106 (Marketing) has no matching employee, so LEFT table columns show NULL.

HOW RIGHT JOIN WORKS

Take all rows from RIGHT table.
Add matching rows from LEFT table.
If no match, fill LEFT table columns with NULL.

JOIN KEYWORD REFERENCE

Keyword	Purpose
RIGHT JOIN	Returns all rows from RIGHT table with matches from LEFT
ON	Specifies the join condition
=	Used to match columns
AS	Gives alias (short name) to tables/columns

VISUAL REPRESENTATION

LEFT TABLE (employees)

id	name	dept_id
1	Aman	101
2	Priya	102
3	Rahul	103
4	Neha	104
5	Vikram	105

RIGHT JOIN ON l.dept_id = d.dept_id

RIGHT TABLE (departments)

dept_id	dept_name
101	Sales
102	HR
103	IT
106	Marketing

⟷ = Match Found · ⇢ = No Match (NULL)

PRACTICAL EXAMPLE

Find all departments and the employees in them.

```
SELECT e.name AS employee_name,
       d.dept_name, d.location
FROM employees e
RIGHT JOIN departments d
ON e.dept_id = d.dept_id;
```

Useful for reports where department information should not be missed.

COMMON USE CASES

- List all departments, along with employees (if any).
- Show all categories or items with related data from another table.
- Generate complete reports from the RIGHT side (main) table.

PRO TIP

Use RIGHT JOIN when the RIGHT table contains the complete set of records you need.

QUICK SUMMARY

Join Type	Tables Involved	Returns	No Match	Join Condition	Relationship	Best For	Result
RIGHT JOIN	Two Tables	All RIGHT rows + Matching LEFT rows	NULL values for LEFT table	Defined using ON clause	Based on related columns	Reports, Listings, Master Data	Complete RIGHT data with related info

MySQL

SELF JOIN

SAME TABLE · RELATIONSHIP · HIERARCHY · QUERY · RESULTS

Join a Table With Itself to Find Relationships Within the Same Data

WHAT IS SELF JOIN?

A SELF JOIN is a regular join, but the table is joined with itself.
It is useful for hierarchical or parent-child relationship data.

SYNTAX

```
SELECT column_list
FROM  table_name t1
JOIN table_name t2
ON t1.common_column = t2.common_column
WHERE condition;
```

SYNTAX BREAKDOWN

SELECT	: Columns to display
FROM	: Table (used twice with aliases)
JOIN	: Join the same table again
ON	: Join condition between aliases
WHERE	: Filter results (optional)

SAMPLE TABLE: *employees*

id	name	manager_id	position
1	CEO	NULL	CEO
2	Alice	1	Manager
3	Bob	1	Manager
4	Charlie	2	Developer
5	David	2	Developer
6	Eva	3	Developer
7	Frank	3	Developer

RESULT (Employee → Manager)

employee	position	manager	position
Alice	Manager	CEO	CEO
Bob	Manager	CEO	CEO
Charlie	Developer	Alice	Manager
David	Developer	Alice	Manager
Eva	Developer	Bob	Manager
Frank	Developer	Bob	Manager

KEY POINTS

- Joins a table with itself using aliases.
- Useful for hierarchical data, parent-child relationships.
- Aliases are mandatory in SELF JOIN.
- Common in Employee-Manager structures.
- Helps in building reports, trees and hierarchy.

EXAMPLE QUERY & RESULT

Example: Employee and Their Manager

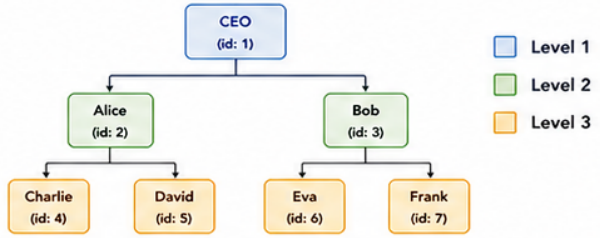
```
SELECT e.name AS employee,
       e.position AS employee_position,
       m.name AS manager,
       m.position AS manager_position
FROM employees e
LEFT JOIN employees m
ON e.manager_id = m.id
ORDER BY e.id;
```

How It Works

Result (All Employees with Their Manager)

employee	employee_position	manager	manager_position
CEO	CEO	NULL	NULL
Alice	Manager	CEO	CEO
Bob	Manager	CEO	CEO
Charlie	Developer	Alice	Manager
David	Developer	Alice	Manager
Eva	Developer	Bob	Manager
Frank	Developer	Bob	Manager

VISUAL HIERARCHY



ALIAS EXPLANATION

Alias	Represents	Used For
e	Employee (current row)	Employee details
m	Manager (related row)	Manager details

Use meaningful aliases for better readability.

ANOTHER EXAMPLE: Employee Count Under Each Manager

```
SELECT m.name AS manager, COUNT(e.id) AS total_employees
FROM employees m
LEFT JOIN employees e ON m.id = e.manager_id
GROUP BY m.id, m.name;
```

manager	total_employees
CEO	2
Alice	2
Bob	2

COMMON USE CASES

- Employee-Manager hierarchy
- Category-Subcategory structure
- Organizational reporting
- Tree or nested relationships
- Finding levels in hierarchy

PRO TIPS

- Always use aliases in SELF JOIN.
- Use LEFT JOIN if you want all rows from the main alias.
- Use INNER JOIN if you want only matching relationships.
- Useful for unlimited levels with recursive queries (CTE).

QUICK SUMMARY

Aspect	Details
Joins	Same table with itself
Aliases	Required
Condition	Relates rows within the table
Returns	Related / Hierarchical data
Use	Hierarchy, Reporting, Trees
Example	Employee and their Manager

SELF JOIN PROCESS

1. Same Table Used Twice → e, m → 3. Define Relation Using ON Clause → 4. Execute Query → 5. Get Related Results

BASIC SELF JOIN TEMPLATE

```
SELECT e.column, m.column
FROM table_name e
JOIN table_name m ON e.common_id = m.common_id;
```

REMEMBER

SELF JOIN helps you find relationships inside the same table!

CROSS JOIN

MySQL

Combine Every Row From One Table With Every Row From Another Table

TABLES · CARTESIAN PRODUCT · RELATIONSHIPS · QUERY · RESULT SET

WHAT IS CROSS JOIN?

CROSS JOIN returns the Cartesian Product of two tables. It combines every row of the first table with every row of the second table.

If table A has m rows and table B has n rows, the result will have m × n rows.

SYNTAX

```
SELECT column_list
FROM table1
CROSS JOIN table2;
```

SYNTAX BREAKDOWN

- `SELECT` : Columns to display
- `FROM` : First table
- `CROSS JOIN` : Combines with every row of the second table (table2)
- Result : Cartesian Product (All combinations)

SAMPLE TABLES & RESULT SET

Table: products

id	product
1	Laptop
2	Mobile
3	Tablet

×

Table: colors

id	color
1	Red
2	Blue

→

Result Set (CROSS JOIN)

product_id	product	color_id	color
1	Laptop	1	Red
1	Laptop	2	Blue
2	Mobile	1	Red
2	Mobile	2	Blue
3	Tablet	1	Red
3	Tablet	2	Blue

All 3 rows from products are combined with all 2 rows from colors.
Total rows = 3 × 2 = 6

KEY POINTS

- Returns the Cartesian Product of two tables.
- All possible combinations of rows are returned.
- No condition (ON or WHERE) is required.
- Result size = (rows in table1) × (rows in table2).
- Use with caution — large tables can produce very big result sets.
- Useful for generating combinations, test data, matrix reports, etc.

EXAMPLE QUERY & EXPLANATION

Example: CROSS JOIN

```
SELECT p.id AS product_id, p.product,
       c.id AS color_id, c.color
FROM products p
CROSS JOIN colors c;
```

How it Works

1. Take first row from products (Laptop)
2. Combine with all rows from colors (Red, Blue)
3. Take second row from products (Mobile)
4. Combine with all rows from colors (Red, Blue)
5. Take third row from products (Tablet)
6. Combine with all rows from colors (Red, Blue)

VISUAL REPRESENTATION

Every row from Table A is paired with every row from Table B

COMMON USE CASES

- Generate all possible combinations.
- Create test data / sample datasets.
- Matrix reports and pivot-like outputs.
- E-commerce product × category views.
- Calendar × Employees scheduling.

PRO TIP

Always apply filters (WHERE clause) after CROSS JOIN to reduce large result sets for better performance.

JOIN KEYWORD REFERENCE

Keyword	Purpose
CROSS JOIN	Returns Cartesian Product (all combinations)
INNER JOIN	Returns matching rows only
LEFT JOIN	Returns all rows from left table + matching from right
RIGHT JOIN	Returns all rows from right table + matching from left
SELF JOIN	Joins a table with itself

ROW COUNT EXAMPLE

Final Result Rows = Rows in Table A × Rows in Table B

PERFORMANCE NOTE

CROSS JOIN can produce very large result sets.

Ensure both tables are small or use filters after the join.

QUICK SUMMARY

Aspect	Details
Join Type	CROSS JOIN
Also Known As	Cartesian Product
Condition	No condition required
Rows Returned	m × n (All combinations)
Use	Combinations, test data, reports, matrices
Risk	Large result set, high memory use

FLOW OF CROSS JOIN

SAMPLE SQL

```
SELECT p.id AS product_id, p.product,
       c.id AS color_id, c.color
FROM products p
CROSS JOIN colors c;
```

UNION & UNION ALL

Combine Results of Multiple SELECT Queries

MULTIPLE SELECTS | MERGE RESULTS | REMOVE DUPLICATES | COMBINE TABLES | FINAL OUTPUT

WHAT IS UNION?

UNION combines the result-set of two or more SELECT statements into a single result-set.

By default, UNION removes duplicate rows from the final result.

UNION Result
(Without Duplicates)

SYNTAX

```
-- UNION (removes duplicates)
SELECT column_list
FROM table1
WHERE condition

UNION

SELECT column_list
FROM table2
WHERE condition;
```

```
-- UNION ALL (keeps duplicates)
SELECT column_list
FROM table1
WHERE condition

UNION ALL

SELECT column_list
FROM table2
WHERE condition;
```

KEY DIFFERENCE

UNION	UNION ALL
Removes duplicate rows	Keeps duplicate rows
Slightly slower (extra processing)	Faster (no duplicate check)
Use when unique results are needed	Use when all rows are needed

SAMPLE TABLES

Table: employees_part_time

id	name	department
1	Aman	Sales
2	Priya	HR
3	Rahul	IT
4	Neha	Finance

Table: employees_full_time

id	name	department
3	Rahul	IT
4	Neha	Finance
5	Vikram	Marketing
6	Pooja	Sales

EXAMPLE QUERY

```
-- Using UNION
SELECT id, name, department
FROM employees_part_time

UNION

SELECT id, name, department
FROM employees_full_time;
```

```
-- Using UNION ALL
SELECT id, name, department
FROM employees_part_time

UNION ALL

SELECT id, name, department
FROM employees_full_time;
```

RESULT COMPARISON

Result using UNION (Remove Duplicates)

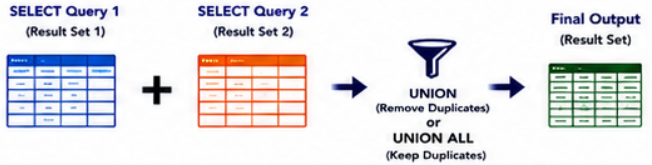
id	name	department
1	Aman	Sales
2	Priya	HR
3	Rahul	IT
4	Neha	Finance
5	Vikram	Marketing
6	Pooja	Sales

UNION ALL includes all rows from both queries, including duplicates.

Result using UNION ALL (Keep Duplicates)

id	name	department
1	Aman	Sales
2	Priya	HR
3	Rahul	IT
4	Neha	Finance
3	Rahul	IT
4	Neha	Finance
5	Vikram	Marketing
6	Pooja	Sales

HOW IT WORKS



- Both SELECT statements must have the same number of columns.
- Corresponding columns must have compatible data types.
- Column names in the final output are taken from the first SELECT.

RULES & NOTES

- The number of columns and their order must be same in all SELECT statements.
- Data types of corresponding columns must be compatible.
- Use UNION when you want unique rows.
- Use UNION ALL when you want all rows (including duplicates).
- ORDER BY can be used at the end of the final query only.

UNION VS UNION ALL

Feature	UNION	UNION ALL
Duplicates	Removed	Not Removed
Performance	Slower	Faster
Use Case	Unique Results	All Results
Example	Reports, Lists	Logs, Transactions

PRACTICAL EXAMPLE

Scenario

You want a list of all employees from part-time and full-time tables.

Recommended

Use UNION for unique employee list.

Use UNION ALL for complete listing including duplicates.

PRO TIP

Use UNION ALL for better performance when you don't need to remove duplicates.

WARNING

Using UNION on large datasets may be slow because of duplicate removal (sorting). Use only when necessary!

FLOW DIAGRAM

QUICK SUMMARY

- UNION combines multiple SELECT results.
- UNION removes duplicates.
- UNION ALL keeps all rows including duplicates.
- Both queries must have same columns and compatible types.
- Use UNION ALL for speed, UNION for uniqueness.

SAMPLE SQL

```
SELECT id, name, department FROM employees_part_time
UNION
SELECT id, name, department FROM employees_full_time
ORDER BY name;
```

SUBQUERIES

A Subquery is a Query Inside Another Query

NESTED QUERY · SELECT · WHERE · FROM · IN · EXISTS · RESULT

WHAT IS A SUBQUERY?

A subquery (inner query) is a query nested inside another SQL query (outer query). It is used to retrieve data that will be used by the main query for further filtering or processing.

Outer Query

↓

Subquery (Inner Query)

SYNTAX

```
SELECT column_list
FROM table_name
WHERE column_name operator
    (SELECT column_list
     FROM table_name
     WHERE condition);
```

KEY POINTS

- Subquery is executed first.
- Its result is passed to the outer query.
- Subquery can be used in WHERE, FROM, SELECT, HAVING clauses.

TYPES OF SUBQUERIES

SCALAR SUBQUERY

Returns a single value (one row, one column)

1

Used with =, <, >, <=, >=, <>

MULTI-ROW SUBQUERY

Returns multiple rows (one column)

2

Used with IN, ANY, ALL, EXISTS

CORRELATED SUBQUERY

Depends on the outer query for its values

3

Executed once for each row of outer query

NON-CORRELATED SUBQUERY

Independent of the outer query

4

Executed once and result is reused

SUBQUERY OPERATORS

=	Equal to
<> / !=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
IN	Match any value in the list
NOT IN	Not match any value
EXISTS	Returns true if subquery returns any row
NOT EXISTS	Returns true if subquery returns no rows
ANY	Compare with any value
ALL	Compare with all values

EXAMPLES OF SUBQUERIES

1. SCALAR SUBQUERY (Returns Single Value)

Find employees who earn more than the average salary.

```
SELECT name, salary
FROM employees
WHERE salary >
  (SELECT AVG(salary)
   FROM employees);
```

Result

Employees with salary greater than average salary.

2. IN SUBQUERY (Multiple Rows)

Find employees who belong to departments located in 'Delhi' or 'Mumbai'.

```
SELECT name, department_id
FROM employees
WHERE department_id IN
  (SELECT department_id
   FROM departments
   WHERE location IN ('Delhi', 'Mumbai'));
```

Result

Employees from departments in Delhi or Mumbai.

3. EXISTS SUBQUERY (Check Existence)

Find employees who have at least one project assigned.

```
SELECT name
FROM employees e
WHERE EXISTS
  (SELECT 1
   FROM projects p
   WHERE p.employee_id = e.id);
```

Result

Employees who have at least one matching project.

4. CORRELATED SUBQUERY (Row by Row)

Find employees who earn more than the average salary in their department.

```
SELECT name, department_id, salary
FROM employees e
WHERE salary >
  (SELECT AVG(salary)
   FROM employees
   WHERE department_id = e.department_id);
```

Result

Employees earning more than their department's average.

SAMPLE TABLES

Table: employees

id	name	department_id	salary
1	Aman	10	50000
2	Priya	20	60000
3	Rahul	10	55000
4	Neha	30	45000
5	Vikram	20	70000

Table: departments

department_id	department_name	location
10	IT	Delhi
20	HR	Mumbai
30	Finance	Pune

Table: projects

project_id	project_name	employee_id
101	Website	1
102	App	3
103	Portal	5

WHERE CAN SUBQUERIES BE USED?

WHERE · FROM (Derived Table) · SELECT (List of Columns) · HAVING

HOW IT WORKS

Subquery Executes First → Returns Result Set → Outer Query Uses Result </> → Final Result

QUICK SUMMARY

- Subquery = Query inside another Query.
- Executed before the outer query (in most cases).
- Helps in filtering, comparing and computing data.
- Use IN for multiple values, EXISTS for existence check.
- Keep subqueries optimized for better performance.

PRO TIP

Use subqueries when you need data from the same table or another table for comparison or filtering.

STORED PROCEDURES

Precompiled SQL Code Stored in the Database and Reused Anytime

CREATE PROCEDURE · CALL · PARAMETERS · REUSABLE CODE · AUTOMATION · BUSINESS LOGIC · PERFORMANCE · DATABASE

WHAT ARE STORED PROCEDURES?

A Stored Procedure is a set of SQL statements stored in the database with a name.

- Precompiled and stored in database.
- Reusable code – call multiple times.
- Improves performance and security.
- Encapsulates business logic.
- Supports parameters, control flow, and error handling.

SYNTAX (CREATE & CALL)

1. CREATE PROCEDURE

```
DELIMITER $$
CREATE PROCEDURE procedure_name(
   IN in_param INT,
   OUT out_param VARCHAR(50)
)
BEGIN
   -- SQL statements
END $$
DELIMITER ;
```

2. CALL PROCEDURE

```
CALL procedure_name(value1, @out_param);
-- or
CALL procedure_name(value1, value2);
```

Parameter Types

IN	: Input parameter (default)
OUT	: Output parameter
INOUT	: Input / Output parameter

EXAMPLE: EMPLOYEE BONUS PROCEDURE

Create Procedure

```
DELIMITER $$
CREATE PROCEDURE GiveBonus(
   IN dept_id INT,
   IN bonus_percent DECIMAL(5,2)
)
BEGIN
   UPDATE employees
   SET salary = salary+ (salary * bonus_percent / 100)
   WHERE department_id = dept_id;
END $$
DELIMITER ;
```

Call Procedure

```
CALL GiveBonus(10, 10.00);
-- 10% bonus to department_id = 10
```

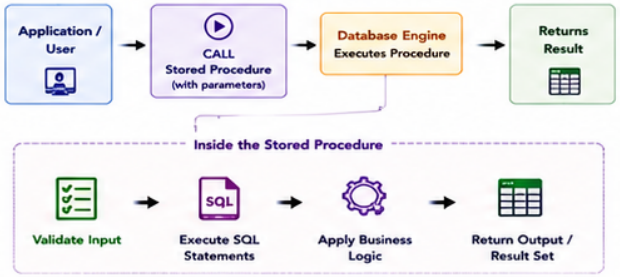
What it does?

Increases salary of all employees in the given department by the specified percentage.

ADVANTAGES

- **Reusable Code** – Write once, use many times.
- **Better Performance** – Precompiled and stored in database.
- **Improved Security** – Restrict direct table access, use procedures instead.
- **Business Logic in Database** – Keeps all business rules in one place.
- **Automation** – Schedule and run complex tasks easily.

HOW IT WORKS (FLOW)



PARAMETERS EXAMPLE

Mode	Example in Procedure	Description	How to Use (CALL)
IN	`IN dept_id INT`	Value passed into procedure	`CALL proc_name(10);`
OUT	`OUT total_emp INT`	Value returned by procedure	`CALL proc_name(@total);` `SELECT @total;`

INOUT	`INOUT count_emp INT`	Value passed in and returned / modified	`SET @c = 5;`
			`CALL proc_name(@c);`
			`SELECT @c;`

SAMPLE TABLE: *employees*

id	name	department_id	salary
1	Aman	10	50000
2	Priya	10	60000
3	Rahul	20	55000
4	Neha	20	45000
5	Vikram	10	70000

After CALL GiveBonus(10, 10.00)

id	name	department_id	salary
1	Aman	10	55000
2	Priya	10	66000
3	Rahul	20	55000
4	Neha	20	45000
5	Vikram	10	77000

COMMON USE CASES

- Data Validation
- Batch Processing
- Complex Business Rules
- Data Synchronization
- Reports & Aggregations
- Audit Logging

PERFORMANCE BENEFITS

BEST PRACTICES

- Use meaningful procedure names.
- Keep procedures small and focused.
- Validate all input parameters.
- Use transactions for data integrity.
- Handle errors using exception handling.
- Document your procedures.
- Test procedures thoroughly.

QUICK SUMMARY

Feature	Stored Procedure
Definition	Precompiled SQL code stored in DB
Execution	CALL procedure_name();
Parameters	IN, OUT, INOUT
Reusability	High
Performance	Better (precompiled)
Security	Better (access control)
Use	Business logic, reports, automation

SAMPLE CREATE & CALL (COMPLETE EXAMPLE)

Create Procedure

```
DELIMITER $$
CREATE PROCEDURE GetEmployeeCount(
   IN dept_id INT,
   OUT total_emp INT
)
BEGIN
SELECT COUNT(*) INTO total_emp FROM employees;
WHERE department_id = dept_id;
END $$
DELIMITER ;
```

Call Procedure

```
DECLARE @total INT;
CALL GetEmployeeCount(10, @total);
SELECT @total AS Total_Employees;
```

Result

Total_Employees
3

REMEMBER

- Stored Procedures make your SQL code modular and powerful.
- They improve performance, security, and maintainability.
- Use them to handle complex logic and automate tasks.

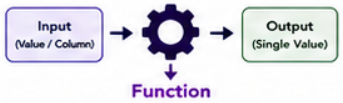
MySQL FUNCTIONS

Built-in Functions to Simplify, Manipulate and Get Better Results

FUNCTIONS · BUILT-IN · PERFORMANCE · EASY TO USE · DATABASE

WHAT ARE FUNCTIONS?

MySQL Functions are predefined formulas that perform operations on data and return a single value.



TYPES OF FUNCTIONS

- String Functions
- Date & Time Functions
- Numeric Functions
- Control Functions
- Miscellaneous Functions

SAMPLE TABLE

Table: users

id	name	age	salary	join_date
1	Aman	25	45000.75	2025-05-20
2	Priya	28	52000.60	2025-05-18
3	Rahul	30	61000.00	2025-05-21
4	Neha	NULL	NULL	NULL

MOST USED MySQL FUNCTIONS WITH EXAMPLES

1 CONCAT()

Joins two or more strings into one string.

Syntax:
```
CONCAT(str1, str2, ... )
```
Example:
```
SELECT CONCAT('Code', 'Sardar')
AS Result;
```
Result: CodeSardar

2 NOW()

Returns the current date and time.

Syntax:
```
NOW()
```
Example:
```
SELECT NOW() AS CurrentDateTime;
```
Result: 2025-05-21 14:35:27

3 CURDATE()

Returns the current date.

Syntax:
```
CURDATE()
```
Example:
```
SELECT CURDATE() AS Today;
```
Result: 2025-05-21

4 ROUND()

Rounds a number to the specified decimal places.

Syntax:
```
ROUND(number, decimals)
```
Example:
```
SELECT ROUND(123.4567, 2)
AS RoundedValue;
```
Result: 123.46

5 UPPER()

Converts a string to uppercase.

Syntax:
```
UPPER(string)
```
Example:
```
SELECT UPPER('codesardar')
AS UpperCase;
```
Result: CODESARDAR

6 LOWER()

Converts a string to lowercase.

Syntax:
```
LOWER(string)
```
Example:
```
SELECT LOWER('CodeSardar')
AS LowerCase;
```
Result: codesardar

7 IFNULL()

Returns a specified value if the expression is NULL.

Syntax:
```
IFNULL(expression, value_if_null)
```
Example:
```
SELECT IFNULL(NULL, 'No Value')
AS ResultValue;
```
Result: No Value

8 LENGTH()

Returns the length of a string.

Syntax:
```
LENGTH(string)
```
Example:
```
SELECT LENGTH('CodeSardar')
AS LengthValue;
```
Result: 10

MORE FUNCTIONS YOU SHOULD KNOW

- LTRIM(str) - Removes leading spaces
- RTRIM(str) - Removes trailing spaces
- TRIM(str) - Removes leading & trailing spaces
- SUBSTRING(str, pos, len) - Extracts substring
- REPLACE(str, from, to) - Replaces substring
- ABS(n) - Absolute value
- CEIL(n) - Rounds up to nearest integer
- FLOOR(n) - Rounds down to nearest integer
- RAND() - Returns a random number
- FORMAT(n, d) - Formats number with d decimals

HOW TO USE FUNCTIONS

```
SELECT FunctionName(column / value)
FROM table_name
WHERE condition;
```

SELECT → FUNCTION() → RESULT

EXAMPLE QUERIES

```
SELECT CONCAT(name, ' | Age: ', age) AS Info FROM users;  -- Using CONCAT()
SELECT NOW() AS CurrentDateTime;                          -- Using NOW()
SELECT CURDATE() AS TodayDate;                            -- Using CURDATE()
SELECT ROUND(salary, 2) AS RoundedSalary FROM users;      -- Using ROUND()
SELECT UPPER(name) AS UpperName FROM users;               -- Using UPPER()
SELECT LOWER(name) AS LowerName FROM users;               -- Using LOWER()
SELECT IFNULL(salary, 0) AS SalaryValue FROM users;       -- Using IFNULL()
SELECT LENGTH(name) AS NameLength FROM users;             -- Using LENGTH()
```

KEY BENEFITS

- Saves Time
- Reduces Manual Work
- Improves Performance
- More Accurate Results
- Easy Data Handling

PRO TIP

Functions make your queries powerful and help in writing clean, readable and efficient SQL code.

TRIGGERS

Automate Actions in MySQL with Triggers

TRIGGER · BEFORE INSERT · AFTER INSERT · UPDATE · DELETE · AUTOMATION · EVENTS · DATABASE

WHAT IS A TRIGGER?

A Trigger is a set of SQL statements that automatically runs (fires) when an event occurs on a table.

Why Use Triggers?

- Enforce business rules
- Maintain audit trails
- Keep data consistent
- Automate tasks
- Reduce manual work

TRIGGER EVENTS

BEFORE INSERT

Executes before a new row is inserted.

Example

```
DELIMITER $$
CREATE TRIGGER before_insert_user
BEFORE INSERT ON users
FOR EACH ROW
BEGIN
   SET NEW.created_at = NOW();
END$$
DELIMITER ;
```

Used to validate or modify data before insertion.

AFTER INSERT

Executes after a new row is inserted.

Example

```
DELIMITER $$
CREATE TRIGGER after_insert_user
AFTER INSERT ON users
FOR EACH ROW
BEGIN
   INSERT INTO user_log(user_id,
          action, action_time)
   VALUES (NEW.id, 'INSERT', NOW());
END$$
DELIMITER ;
```

Used for logging or performing actions after insertion.

BEFORE UPDATE

Executes before a row is updated.

Example

```
DELIMITER $$
CREATE TRIGGER before_update_user
BEFORE UPDATE ON users
FOR EACH ROW
BEGIN
   IF NEW.salary < 0 THEN
      SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Salary
          cannot be negative';
   END IF;
END$$
DELIMITER ;
```

Used to validate or modify data before update.

AFTER DELETE

Executes after a row is deleted.

Example

```
DELIMITER $$
CREATE TRIGGER after_delete_user
AFTER DELETE ON users
FOR EACH ROW
BEGIN
   INSERT INTO user_log(user_id,
          action, action_time)
   VALUES (OLD.id, 'DELETE', NOW());
END$$
DELIMITER ;
```

Used for logging or cleanup after deletion.

SYNTAX

```
DELIMITER $$
CREATE TRIGGER trigger_name
    timing event ON table_name
FOR EACH ROW
BEGIN
    -- SQL statements
END$$
DELIMITER ;
```

KEYWORDS

CREATE TRIGGER	Creates a new trigger
BEFORE / AFTER	When to execute
INSERT	When a row is inserted
UPDATE	When a row is updated
DELETE	When a row is deleted
FOR EACH ROW	Trigger for each affected row
NEW	Refers to new row data
OLD	Refers to old row data
BEGIN ... END	Block of SQL statements

SAMPLE TABLE

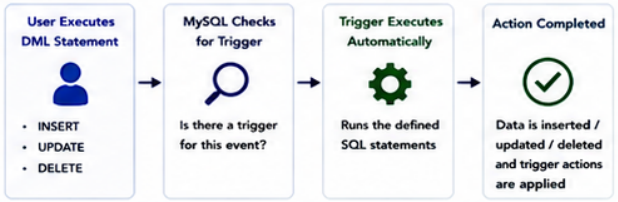
Table: users

id	name	email	salary	created_at
1	Aman	aman@mail.com	50000	2025-05-20 10:00:00
2	Priya	priya@mail.com	60000	2025-05-21 11:00:00
3	Rahul	rahul@mail.com	55000	2025-05-22 12:00:00

Table: user_log (Audit Table)

log_id	user_id	action	action_time
1	1	INSERT	2025-05-20 10:00:01
2	2	INSERT	2025-05-21 11:00:01
3	3	DELETE	2025-05-22 13:30:00

HOW TRIGGERS WORK



EXAMPLE: AUDIT TRAIL WITH TRIGGERS

```
-- After Insert Trigger (Already shown above)
-- After Delete Trigger (already shown above)
-- After Update Trigger Example
DELIMITER $$
CREATE TRIGGER after_update_user
AFTER UPDATE ON users
FOR EACH ROW
BEGIN
   INSERT INTO user_log(user_id, action, action_time)
   VALUES (NEW.id, 'UPDATE', NOW());
END$$
DELIMITER ;
```

Result in user_log table

log_id	user_id	action	action_time
1	1	INSERT	2025-05-20 10:00:01
2	2	INSERT	2025-05-21 11:00:01
3	3	UPDATE	2025-05-22 12:30:01
4	3	DELETE	2025-05-22 13:30:00

TRIGGER EVENTS SUMMARY

Event	BEFORE	AFTER	Use Case
INSERT	✓	Validate / Set defaults	✓ Logging / Notifications
UPDATE	✓	Validate / Modify data	✓ Logging / Track changes
DELETE	–	-	✓ Logging / Cleanup

BENEFITS OF TRIGGERS

- **Automation** — Reduces manual work.
- **Data Integrity** — Enforces business rules.
- **Consistency** — Keeps data accurate.
- **Audit & Logging** — Tracks data changes.
- **Security** — Restrict or validate data.
- **Performance** — Faster than external scripts.

BEST PRACTICES

- Keep triggers small and focused.
- Avoid complex logic inside triggers.
- Use triggers only when necessary.
- Test triggers thoroughly.
- Document trigger purpose and behavior.

IMPORTANT NOTES

- Triggers cannot be called directly using CALL.
- They fire automatically.
- Be careful to avoid infinite loops.
- Use OLD and NEW to access row data.
- Triggers run within the same transaction.

TRANSACTIONS

Manage Multiple Operations as a Single Unit of Work

START TRANSACTION | COMMIT (SAVE) | ROLLBACK (UNDO) | SAVEPOINT (PARTIAL ROLLBACK) | RELIABLE DATA | RECOVERY & SAFETY

WHAT IS A TRANSACTION?

A transaction is a sequence of one or more SQL operations performed as a single logical unit of work.

Either all operations are executed successfully (COMMIT) or none are (ROLLBACK).

TRANSACTION CONTROL STATEMENTS

1. START TRANSACTION

Marks the beginning of a transaction.

Syntax:

```
START TRANSACTION;
or
BEGIN;
```

2. COMMIT

Saves (makes permanent) all changes made in the transaction.

Syntax:

```
COMMIT;
```

3. ROLLBACK

Undoes all changes made in the transaction.

Syntax:

```
ROLLBACK;
```

4. SAVEPOINT

Sets a point within a transaction to which you can later rollback.

Syntax:

```
SAVEPOINT savepoint_name;
```

5. ROLLBACK TO SAVEPOINT

Rolls back the transaction to a previous savepoint.

Syntax:

```
ROLLBACK TO SAVEPOINT
savepoint_name;
```

ACID PROPERTIES

A – Atomicity
All operations in a transaction succeed or all fail.

C – Consistency
Transaction brings the database from one valid state to another.

I – Isolation
Transactions are executed independently without interfering with each other.

D – Durability
Once committed, changes are permanent even in case of system failure.

EXAMPLE: BANK TRANSFER

Transfer ₹500 from Account A to Account B

Initial Balances

Account	Balance (₹)
A	5000
B	3000

STEP 1: START

```
START TRANSACTION;
```

Transaction started.

STEP 2: UPDATE A

```
UPDATE accounts
SET balance = balance - 500
WHERE account_id = 'A';
```

₹500 deducted from Account A.

STEP 3: UPDATE B

```
UPDATE accounts
SET balance = balance + 500
WHERE account_id = 'B';
```

₹500 added to Account B.

STEP 4: COMMIT

```
COMMIT;
```

All changes saved permanently.

STEP 5: RESULT

Final Balances

Account	Balance (₹)
A	4500
B	3500

Transfer Successful! ✅

WHAT IF SOMETHING FAILS?

If any error occurs (e.g., Insufficient funds)

1. START TRANSACTION;
2. UPDATE A (fail) ❌
3. UPDATE B (not executed)
4. ROLLBACK; ↩

Result: No changes saved.

Balances After Rollback

Account	Balance (₹)
A	5000
B	3000

Data is safe! 🛡

WHEN TO USE TRANSACTIONS?

- ✅ Money transfers in banking apps
- ✅ E-commerce order placement
- ✅ Inventory updates
- ✅ Booking systems
- ✅ Any process with multiple SQL operations

> "Use transactions to keep your data reliable and error-free."

HOW TRANSACTIONS WORK

SAVEPOINT EXAMPLE

```
START TRANSACTION;
UPDATE accounts SET balance = balance - 500 WHERE account_id = 'A';
SAVEPOINT after_deduct;
UPDATE accounts SET balance = balance + 500 WHERE account_id = 'B';
-- If any error now
ROLLBACK TO SAVEPOINT after_deduct;   -- Rollback to savepoint
COMMIT;                               -- Commit the valid part
```

KEY BENEFITS

- Maintains Data Integrity
- Handles Errors Gracefully
- Ensures Reliable Data Processing
- Supports Recovery from Failure
- Improves Application Reliability.

QUICK SUMMARY

Command	Purpose
START TRANSACTION	Begin a new transaction
COMMIT	Save all changes
ROLLBACK	Undo all changes
SAVEPOINT name	Set a savepoint
ROLLBACK TO name	Rollback to savepoint

BEST PRACTICES

- Keep transactions short.
- Always COMMIT or ROLLBACK.
- Use meaningful SAVEPOINT names.
- Handle exceptions properly.
- Test rollback scenarios.

REMEMBER

Transactions = Reliability
COMMIT = Save
ROLLBACK = Undo

Think Before You Commit, Rollback When in Doubt!

RELIABLE DATA, STRONG APPLICATIONS.

Transactions are the backbone of data integrity in any application.

COMMIT & ROLLBACK

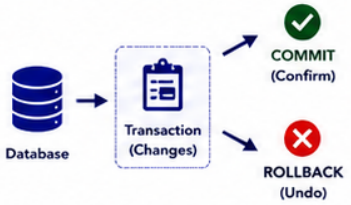
Control Your Transactions, Protect Your Data

COMMIT — Confirm Changes | ROLLBACK — Undo Changes | SAVEPOINT — Set Checkpoint | TRANSACTIONS — Manage Work | SAFETY — Data Protection | RECOVERY — Restore Data

WHAT ARE COMMIT & ROLLBACK?

In MySQL, changes are first made in a transaction.

You can either SAVE (COMMIT) the changes permanently or CANCEL (ROLLBACK) them to keep the data safe.



KEY COMMANDS

1. COMMIT

Saves all the changes made in the transaction permanently.

Syntax:

```
COMMIT;
```

Example:

```
UPDATE users SET balance = balance - 500
WHERE id = 1;
COMMIT;  -- Changes are saved permanently
```

2. ROLLBACK

Undoes all the changes made in the transaction since the last COMMIT.

Syntax:

```
ROLLBACK;
```

Example:

```
UPDATE users SET balance = balance - 500
WHERE id = 1;
ROLLBACK;  -- Changes are undone
```

3. SAVEPOINT

Sets a point within a transaction to which you can later roll back.

Syntax:

```
SAVEPOINT savepoint_name;
ROLLBACK TO SAVEPOINT savepoint_name;
```

Example:

```
SAVEPOINT sp1;   -- Set a savepoint
UPDATE users SET balance = balance - 200 WHERE id = 2;
ROLLBACK TO SAVEPOINT sp1;  -- Rollback to savepoint
```

TRANSACTION FLOW

START TRANSACTION

↓

MAKE CHANGES (INSERT, UPDATE, DELETE)

↓

COMMIT (Save Changes) | ROLLBACK (Undo Changes)

↓

DATABASE

COMMIT vs ROLLBACK

COMMIT	ROLLBACK
Make changes permanent	Undo changes
Data is saved in database	Data is not saved
Cannot be undone	Can be done anytime before COMMIT
Ensures data is consistent	Helps in error recovery

TRANSACTION EXAMPLE

Step	Action	Command	Result
1	Start Transaction	`START TRANSACTION;`	Transaction started
2	Update Data (A)	`UPDATE accounts SET balance = balance - 100 WHERE id = 1;`	₹100 deducted from Account A
3	Update Data (B)	`UPDATE accounts SET balance = balance + 100 WHERE id = 2;`	₹100 added to Account B
4A	Commit	`COMMIT;`	All changes saved permanently
4B	Rollback	`ROLLBACK;`	All changes undone

SAVEPOINT EXAMPLE

```
1  START TRANSACTION;
2  UPDATE accounts SET balance = balance - 500 WHERE id = 1;
3  SAVEPOINT sp1;              -- Savepoint set
4  UPDATE accounts SET balance = balance - 200 WHERE id = 1;
5  -- Oops! Mistake found
6  ROLLBACK TO SAVEPOINT sp1;  -- Go back to savepoint
7  COMMIT;                     -- Confirm correct changes
```

KEY BENEFITS

- Ensures Data Integrity
- Easy Error Recovery
- Safe & Reliable Operations
- Better Control Over Data
- Supports Complex Operations

WHEN TO USE?

- While transferring money
- During bulk data updates
- When multiple operations are related
- To maintain data consistency

BENEFITS SUMMARY

Data Integrity | Error Recovery | Data Safety | Consistency Assured | Better Reliability

BEST PRACTICES

- Always use transactions for critical operations.
- COMMIT only when all operations are successful.
- Use ROLLBACK when any error occurs.
- Use SAVEPOINT for partial rollbacks in large transactions.
- Keep transactions short and simple.

IMPORTANT NOTES

- After COMMIT, changes cannot be undone.
- ROLLBACK works only before COMMIT.
- SAVEPOINT is available only within the current transaction.
- Use transactions to keep your data safe and reliable.

QUICK SYNTAX REFERENCE

```
START TRANSACTION;                -- Begin transaction
COMMIT;                           -- Save changes
ROLLBACK;                         -- Undo all changes
SAVEPOINT sp_name;                -- Set savepoint
ROLLBACK TO SAVEPOINT sp_name;    -- Rollback to savepoint
```

Think Before You Commit, | Commit When Sure, | Rollback When in Doubt! | Safe Data, Strong System.

BACKUP DATABASE

Secure Your Data Today, Restore Anytime

MYSQLDUMP · BACKUP · DATABASE · EXPORT · COMMAND LINE · .SQL FILE · RECOVERY · SECURITY

WHAT IS BACKUP?

A backup is a copy of your database that you can use to restore data in case of loss, corruption, or failure.

Database → Backup → Backup File (.sql)

WHY BACKUP?

- Prevent data loss
- Easy recovery
- Data migration
- Security & safety
- Disaster protection

USING MYSQLDUMP (COMMAND LINE)

mysqldump is a MySQL utility that creates logical backups of databases.

Basic Syntax:

```
mysqldump -u username -p database_name > backup.sql
```

Example:

```
mysqldump -u root -p mydb > mydb_backup.sql
```

Parameter Guide:

`-u username`	: MySQL username
`-p`	: Prompt for password
`database_name`	: Name of the database to backup
`> backup.sql`	: Output backup to .sql file

Command Line

```
mysql> mysqldump -u root -p mydb > mydb_backup.sql
Enter password: *******
-- Dumping database structure
-- Dumping data for table users
-- Dump completed successfully
```

mydb → mydb_backup.sql (Backup File)

EXPORT OPTIONS

Full Database Backup
Backup entire database (structure + data)

```
mysqldump -u root -p mydb > mydb_backup.sql
```

Specific Tables Backup
Backup selected table(s)

```
mysqldump -u root -p mydb table1 table2 > tables.sql
```

Structure Only (No Data)
Export only table structure

```
mysqldump -u root -p mydb --no-data > structure.sql
```

Data Only (No Structure)
Export only data

```
mysqldump -u root -p mydb --no-create-info > data.sql
```

PRACTICAL EXAMPLE

1. Open Command Line — `C:\>_`
2. Run mysqldump Command

```
C:\> mysqldump -u root -p mydb
  > mydb_backup.sql
```

3. Enter Password

```
Enter password: ******
```

4. Backup Created — mydb_backup.sql (Backup File)

.SQL FILE CONTENT (EXAMPLE)

```
-- MySQL dump 10.13  Distrib 8.0.x
-- Host: localhost    Database: mydb
-- ------------------------------------------------------

CREATE DATABASE IF NOT EXISTS `mydb`;
USE `mydb`;

-- Table structure for table `users`
CREATE TABLE `users` (
  `id` int NOT NULL AUTO_INCREMENT,
  `name` varchar(50) NOT NULL,
  `email` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

-- Dumping data for table `users`
INSERT INTO `users` (`id`, `name`, `email`) VALUES
(1, 'Aman', 'aman@gmail.com'),
(2, 'Priya', 'priya@gmail.com');

-- Dumpg completed on 2025-05-20 12:30:45
```

RECOVERY (RESTORE) FROM BACKUP

To restore a database from .sql file:

```
mysql -u root -p mydb < mydb_backup.sql
```

Example:

```
mysql> mysql -u root -p mydb < mydb_backup.sql
Enter password: *******
Query OK, 0 rows affected (0.12 sec)
Database restored successfully!
```

BACKUP FILE LOCATION

Backup file will be saved in the current directory.

C:\mysql_backups\ → mydb_backup.sql (Backup File)

Tip: Use absolute path to save backup at any location.

```
mysqldump -u root -p mydb > C:\backup\mydb_backup.sql
```

BENEFITS OF BACKUP

- Quick recovery in data loss
- Migrate data to another server
- Test or development purpose
- Disaster recovery plan
- Peace of mind & data safety

SECURITY TIPS

- Store backups in a secure location
- Restrict access to backup files
- Use strong passwords
- Encrypt sensitive backups
- Delete old backups regularly

BEST PRACTICES

- Schedule Regular Backups
- Store in Multiple Locations
- Verify Backup Files
- Keep Backup Secure
- Test Restore Periodically
- Remove Old Backups

QUICK SUMMARY

Tool	: mysqldump
Command	: mysqldump -u user -p db > file.sql
Output File	: .sql file (Backup)
Restore Command	: mysql -u user -p db < file.sql
Use Case	: Backup, Export, Migration, Recovery

Backup Today, Secure Tomorrow!

RESTORE DATABASE

Recover Your Data Quickly and Safely

RESTORE Import Data | IMPORT .sql File | MYSQL COMMAND | .SQL FILE Backup File | RECOVERY Get Data Back | DATABASE Your Data | OVERWRITE Replace Data | SAFETY Secure Restore

WHAT IS RESTORE?

Restore is the process of importing a backup file (.sql) back into MySQL to recover your database.

WHY RESTORE?

- Data loss recovery
- Server crash recovery
- Migrate to new server
- Accidental deletion fix
- Disaster recovery

RESTORE DATABASE USING MYSQL COMMAND

Basic Syntax

```
mysql -u username -p database_name < backup_file.sql
```

Syntax Explained

`-u username`	: MySQL username
`-p`	: Prompt for password
`database_name`	: Name of the database to restore into
`< backup_file.sql`	: Backup .sql file to import

Example

```
mysql -u root -p mydb < mydb_backup.sql
```

RESTORE PROCESS

Tip: Always take a fresh backup before restoring!

IMPORT OPTIONS

Import Entire Database
Restore full database from .sql file

```
mysql -u root -p mydb < backup.sql
```

Import Specific Database
Restore into a specific database

```
mysql -u root -p target_db < backup.sql
```

Import with Overwrite
Drop and recreate tables (use with caution)

```
mysql -u root -p target_db < backup.sql
```

Import with No Data (Structure Only)
Restore only structure, no data

```
mysql -u root -p target_db < backup.sql -d
```

BEFORE YOU RESTORE

- Make sure you have a valid .sql backup file
- Note your database name and login details
- Close all applications using the database
- Backup current data (just in case)
- Ensure you have enough disk space

STEP-BY-STEP: RESTORE DATABASE

SAMPLE SCENARIOS

Restore on same server

```
mysql -u root -p mydb < backup.sql
```

Restore on different server

```
mysql -u root -p mydb < backup.sql
```

Overwrite existing data

```
mysql -u root -p mydb < backup.sql
```

EXAMPLE: RESTORE PROCESS

1. Open Command Line →
2. Run Restore Command →
3. Enter Password →
4. Processing... →
5. Success Message →

```
C:\> mysql -u root -p mydb < mydb_backup.sql
Enter password: *******
-- Checking file: mydb_backup.sql
-- Importing database: mydb
-- Please wait...
Query OK, 0 rows affected (1.23 sec)
Database restored successfully!
```

COMMON FILE LOCATIONS

Windows	: C:\backups\backup.sql
Linux	: /home/user/backups/backup.sql
macOS	: ~/backups/backup.sql

Note: Use full path if file is not in current directory.

TIPS & BEST PRACTICES

- Always verify the backup file before restoring.
- Restore on a test database first.
- Do not restore on a live database without backup.
- Check file size and integrity.
- Use transactions for large imports if possible.

POTENTIAL RISKS

- Existing data may be overwritten.
- Incompatible MySQL versions may cause errors.
- Large files may take time to import.
- Database may become unavailable during restore.

QUICK SUMMARY

mysql Command | Import Backup | .sql File Inpore | Recovery Complete | Safe & Secure

Backup Today, Restore Tomorrow!

Quick Restore Command

```
mysql -u root -p database_name < backup.sql
```

Recover Fast. Protect Always.

MySQL IMPORT & EXPORT

Move Data Easily, Safely and Efficiently

IMPORT Bring Data In | EXPORT Send Data Out | CSV FILE Comma Separated | SQL FILE Structured Data | WORKBENCH GUI Tool | DATA TRANSFER Move Data | MIGRATION Migrate Data | BACKUP Protect Data

WHAT IS IMPORT & EXPORT?

Import is used to bring data into MySQL from files or other databases.

Export is used to save data from MySQL to files for backup, sharing, or migration.

IMPORT (Bring In) → MySQL Database → EXPORT (Send Out)

COMMON USES

- Backup your database
- Migrate data to another server
- Move data between environments
- Share data with others
- Archive data for reporting

IMPORT DATA

1. Import from SQL File

```
mysql -u username -p database_name < file.sql
```

2. Import from CSV File

```
LOAD DATA INFILE 'data.csv'
INTO TABLE table_name
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;
```

Supported Formats

- SQL File (*.sql)
- CSV File (*.csv)
- Excel File (*.xlsx)
- JSON File (*.json)
- XML File (*.xml)

Import Sources

SQL File | CSV File | Excel File | JSON File | XML File | Other DB

EXPORT DATA

1. Export to SQL File (Backup)

```
mysqldump -u username -p database_name > backup.sql
```

2. Export to CSV File

```
SELECT * FROM table_name
INTO OUTFILE 'data.csv'
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n';
```

Export Options

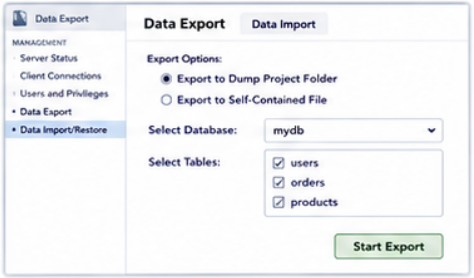
- Structure (only table structure)
- Data (only table data)
- Structure + Data (full backup)
- Specific Tables
- Custom Queries

Export Destinations

SQL File | CSV File | Excel File | TXT File | JSON File | Other DB

USING MYSQL WORKBENCH

1. Open MySQL Workbench
2. Go to Server → Data Export (or Data Import)
3. Select Database or Tables
4. Choose Export/Import Destination
5. Configure Options
6. Start Export/Import



IMPORT vs EXPORT

IMPORT (Bring Data In)	EXPORT (Send Data Out)
Loads data into MySQL	Saves data from MySQL
Use for migration or restore	Use for backup or sharing
Sources: SQL, CSV, Excel, etc.	Destinations: SQL, CSV, Text, etc.
Command: LOAD DATA, mysql < file.sql	Command: mysqldump, SELECT INTO OUTFILE

VS

DATA TRANSFER & MIGRATION

Source Database → Extract (Data Export) → Transfer (File/Network) → Import (Target Server) → Target Database

Use Case: Moving data from Local to Server, or Server to Server.

EXAMPLE COMMANDS

```
-- Export entire database
mysqldump -u root -p mydb > mydb_backup.sql

-- Export specific table
mysqldump -u root -p mydb users > users_backup.sql

-- Export with structure only
mysqldump -u root -p mydb --no-data > structure.sql

-- Import SQL file
mysql -u root -p mydb < backup.sql

-- Import CSV file into table
LOAD DATA INFILE 'data.csv' INTO TABLE users
FIELDS TERMINATED BY ',' ENCLOSED BY '"' LINES TERMINATED BY '\n' IGNORE 1 ROWS;
```

FILE TYPES & EXTENSIONS

SQL File	.sql	Full backup / Structure + Data
CSV File	.csv	Comma Separated Values
Excel File	.xlsx	Microsoft Excel Format
JSON File	.json	JavaScript Object Notation
XML File	.xml	eXtensible Markup Language
Text File	.txt	Plain Text Format

BACKUP

Export your database regularly to prevent data loss.

MIGRATION

Move data between servers or environments seamlessly.

DATA SECURITY

Keep backups in secure locations and restrict access.

AUTOMATION

Schedule exports using cron jobs or Task Scheduler.

SCALABILITY

Easily transfer large datasets with proper tools and methods.

BEST PRACTICES

- Always backup before making important changes.
- Verify data after import/export.
- Use compressed backups for large databases.
- Store backups in secure and redundant locations.
- Test restore process regularly.

QUICK SUMMARY

Action	Tool / Command
Export (Backup)	mysqldump
Import (Restore)	mysql / LOAD DATA
Export CSV	SELECT INTO OUTFILE
Import CSV	LOAD DATA INFILE
GUI Tool	MySQL Workbench

★ **IMPORT** to bring data in. **EXPORT** to send data out. **BACKUP** to stay safe. **MIGRATE** to grow. | Good Data Today, Better Tomorrow!

USER MANAGEMENT

Create, Manage and Secure Database Users

CREATE USER | ALTER USER | DROP USER | PASSWORD | LOGIN | SECURITY | DATABASE USER

WHAT IS USER MANAGEMENT?

User Management in MySQL allows you to create, modify, and delete user accounts and control their access privileges for databases.

KEY BENEFITS

- Secure your database
- Control access and permissions
- Manage user accounts easily
- Track and manage logins
- Improve authentication & security

1. CREATE USER

Create a new user and set password.

```
CREATE USER 'username'@'host'
IDENTIFIED BY 'password';
```

Example:

```
CREATE USER 'jeet'@'localhost'
IDENTIFIED BY 'StrongPass@123';
```

Common Hosts:

- `'localhost'` – Local access only
- `'%'` – Access from anywhere
- `'192.168.%'` – Access from IP range

2. ALTER USER

Modify user password or properties.

```
ALTER USER 'username'@'host'
IDENTIFIED BY 'new_password';
```

Example:

```
ALTER USER 'jeet'@'localhost'
IDENTIFIED BY 'NewPass@@456';
```

Other Alter Options:

```
ALTER USER 'u'@'h' ACCOUNT LOCK;
-- Lock user
ALTER USER 'u'@'h' ACCOUNT UNLOCK;
-- Unlock user
```

3. DROP USER

Remove a user from the system.

```
DROP USER 'username'@'host';
```

Example:

```
DROP USER 'jeet'@'localhost';
```

Dropping a user removes all privileges granted to that user.

AUTHENTICATION & LOGIN

Users login using their credentials.

```
mysql -u username -p
Enter password: ********
```

Authentication Plugins:

- `mysql_native_password` (Default)
- `caching_sha2_password` (MySQL 8.0+)

USER PRIVILEGES OVERVIEW

Grant privileges to users for databases and objects.

```
GRANT privilege_list ON database_name.table_name
TO 'username'@'host';
```

Example:

```
GRANT ALL PRIVILEGES ON mydb.*
TO 'jeet'@'localhost';
```

Common Privileges:

SELECT	– Read data	CREATE	– Create objects
INSERT	– Insert data	ALTER	– Modify objects
UPDATE	– Modify data	DROP	– Drop objects
DELETE	– Delete data	ALL PRIVILEGES	– All access

USER MANAGEMENT WORKFLOW

Important Commands

```
-- Show all users
SELECT User, Host FROM mysql.user;
-- Show grants for a user
SHOW GRANTS FOR 'username'@'host';
-- Flush privileges (apply changes)
FLUSH PRIVILEGES;
```

EXAMPLE: COMPLETE USER MANAGEMENT

```
1  -- Create User
   CREATE USER 'student'@'%' IDENTIFIED BY 'Stud@123';
2  -- Grant Privileges
   GRANT SELECT, INSERT, UPDATE ON school.* TO 'student'@'%';
3  -- Apply Changes
   FLUSH PRIVILEGES;
4  -- Alter Password
   ALTER USER 'student'@'%' IDENTIFIED BY 'NewStud@456';
5  -- Drop User (if needed)
   DROP USER 'student'@'%';
```

SECURITY BEST PRACTICES

- Use strong passwords
- Grant only required privileges
- Remove unused users
- Lock users when not in use
- Keep MySQL updated

PASSWORD TIPS

- Use letters, numbers & symbols
- Minimum 8-12 characters
- Avoid dictionary words
- Change passwords regularly

COMMON MISTAKES

- Giving ALL privileges to all users
- Using weak passwords
- Not revoking unused access
- Not locking inactive users

DATABASE USER TYPES

- Administrators (Full Access)
- Developers (Limited Access)
- Read-Only Users
- Application Users

QUICK SUMMARY

Action	Command
Create User	`CREATE USER 'user'@'host' IDENTIFIED BY 'pass';`
Alter User	`ALTER USER 'user'@'host' IDENTIFIED BY 'new_pass';`
Drop User	`DROP USER 'user'@'host';`
Show Users	`SELECT User, Host FROM mysql.user;`
Show Grants	`SHOW GRANTS FOR 'user'@'host';`

Strong Users • Strong Passwords • Strong Security | Manage Users Smartly. Secure Your Database. | Security Today, Safety Forever!

PERMISSIONS

Control Access. Protect Data. Ensure Security.

GRANT — Give Access | REVOKE — Remove Access | PRIVILEGES — What Users Can Do | ROLES — Group of Privileges | READ — View Data | WRITE — Modify Data | ADMIN — Full Control | SECURITY — Secure Database

WHAT ARE PERMISSIONS?

Permissions in MySQL determine what users can do on databases, tables, and objects.

They are assigned using GRANT and removed using REVOKE.

KEY BENEFITS

- Restrict unauthorized access
- Protect sensitive data
- Control actions (Read, Write, Admin)
- Follow least privilege principle
- Enhance database security

1. GRANT – GIVE PRIVILEGES

Grant privileges to users on databases, tables or objects.

```
GRANT privilege_list ON database.table
TO 'username'@'host' [IDENTIFIED BY 'password'];
```

Example:

```
GRANT SELECT, INSERT, UPDATE ON school.*
TO 'jeet'@'localhost' IDENTIFIED BY 'StrongPass@123';
```

Common Options:

- `GRANT ALL PRIVILEGES` – All privileges
- `GRANT SELECT` – Read data
- `GRANT INSERT, UPDATE, DELETE` – Modify data
- `GRANT CREATE, DROP` – Create/Drop objects
- `GRANT INDEX, ALTER` – Advanced operations

2. REVOKE – REMOVE PRIVILEGES

Remove previously granted privileges.

```
REVOKE privilege_list ON database.table
FROM 'username'@'host';
```

Example:

```
REVOKE INSERT, UPDATE ON school.*
FROM 'jeet'@'localhost';
```

Notes:

- `REVOKE` removes specific privileges.
- `REVOKE ALL PRIVILEGES` removes all.
- Does not drop the user from the system.

3. COMMON PRIVILEGES

Privilege	Description	Type
SELECT	Read data	Read
INSERT	Insert new data	Write
UPDATE	Modify existing data	Write
DELETE	Delete data	Write
CREATE	Create databases/tables	Admin
DROP	Drop databases/tables	Admin
ALTER	Modify structure	Admin
INDEX	Create/Drop indexes	Admin
EXECUTE	Execute stored routines	Admin
ALL PRIVILEGES	All available privileges	Admin

4. ROLES – GROUP OF PRIVILEGES

Roles help manage permissions easily.

```
CREATE ROLE 'read_only';
GRANT SELECT ON mydb.* TO 'read_only';
GRANT 'read_only' TO 'jeet'@'localhost';
```

Why Use Roles?

- Simplify permission management
- Assign/Remove permissions quickly
- Reuse roles for multiple users
- Improve security & organization

5. PERMISSION LEVELS (SCOPE)

Level	Scope	Example	Description
Global	All databases	`GRANT SELECT ON *.*`	Affects the entire server
Database	Specific database	`GRANT SELECT ON mydb.*`	Affects all tables in a database
Table	Specific table	`GRANT SELECT ON mydb.users`	Affects one table
Column	Specific column	`GRANT SELECT (name) ON mydb.users`	Affects specific column(s)
Routine	Stored procedure/function	`GRANT EXECUTE ON PROCEDURE mydb.proc1`	Affects stored routines

6. PRIVILEGE MATRIX (EXAMPLES)

Role / User	SELECT	INSERT	UPDATE	DELETE	ADMIN
Read Only User	✓	–	–	–	–
Data Entry User	✓	✓	–	–	–
Editor User	✓	✓	✓	–	–
Admin User	✓	✓	✓	✓	✓
Database Admin	✓	✓	✓	✓	✓

7. EXAMPLE WORKFLOW

1 Create User
```
CREATE USER 'jeet'@'localhost' IDENTIFIED BY 'StrongPass@123';
```
↓
2 Grant Privileges
```
GRANT SELECT, INSERT ON school.* TO 'jeet'@'localhost';
```
↓
3 User Works
User can read and insert data.
↓
4 Revoke Privileges (if needed)
```
REVOKE INSERT ON school.* FROM 'jeet'@'localhost';
```
↓
5 Drop User (if needed)
```
DROP USER 'jeet'@'localhost';
```

8. AUTHENTICATION & LOGIN

Users login using their credentials.

```
mysql -u jeet -p
Enter password: ********
```

Authentication Plugins:

- mysql_native_password (Default)
- caching_sha2_password (MySQL 8.0+)
- sha256_password
- auth_socket (Linux)

9. BEST PRACTICES

- Use least privilege – grant only what is needed.
- Use roles for easier management.
- Change passwords regularly.
- Remove unused users and privileges.
- Audit user permissions periodically.

10. QUICK COMMAND REFERENCE

Create User	`CREATE USER 'user'@'host' IDENTIFIED BY 'pass';`
Grant Privileges	`GRANT SELECT ON mydb.* TO 'user'@'host';`
Revoke Privileges	`REVOKE SELECT ON mydb.* FROM 'user'@'host';`
Show Grants	`SHOW GRANTS FOR 'user'@'host';`
Drop User	`DROP USER 'user'@'host';`
Flush Privileges	`FLUSH PRIVILEGES;`

11. PERMISSIONS SUMMARY

Action	Command	Purpose
GRANT	`GRANT privilege ON object TO user;`	Give access / privileges
REVOKE	`REVOKE privilege ON object FROM user;`	Remove access / privileges
SHOW GRANTS	`SHOW GRANTS FOR 'user'@'host';`	View user privileges
ROLES	`CREATE ROLE, GRANT role TO user;`	Assign group of privileges
SECURITY	Manage users, roles and privileges	Keep database secure

Secure Access Today, | Control Permissions Wisely, | Protect Data Always!

MySQL WORKBENCH

Visual Design. SQL Development. Database Management.

WORKBENCH | SQL EDITOR | QUERY | ER DIAGRAM | DATA EXPORT | DATA IMPORT | USERS | ADMINISTRATION

WHAT IS MYSQL WORKBENCH?

MySQL Workbench is a visual tool for database architects, developers and DBAs. It provides a unified environment for:

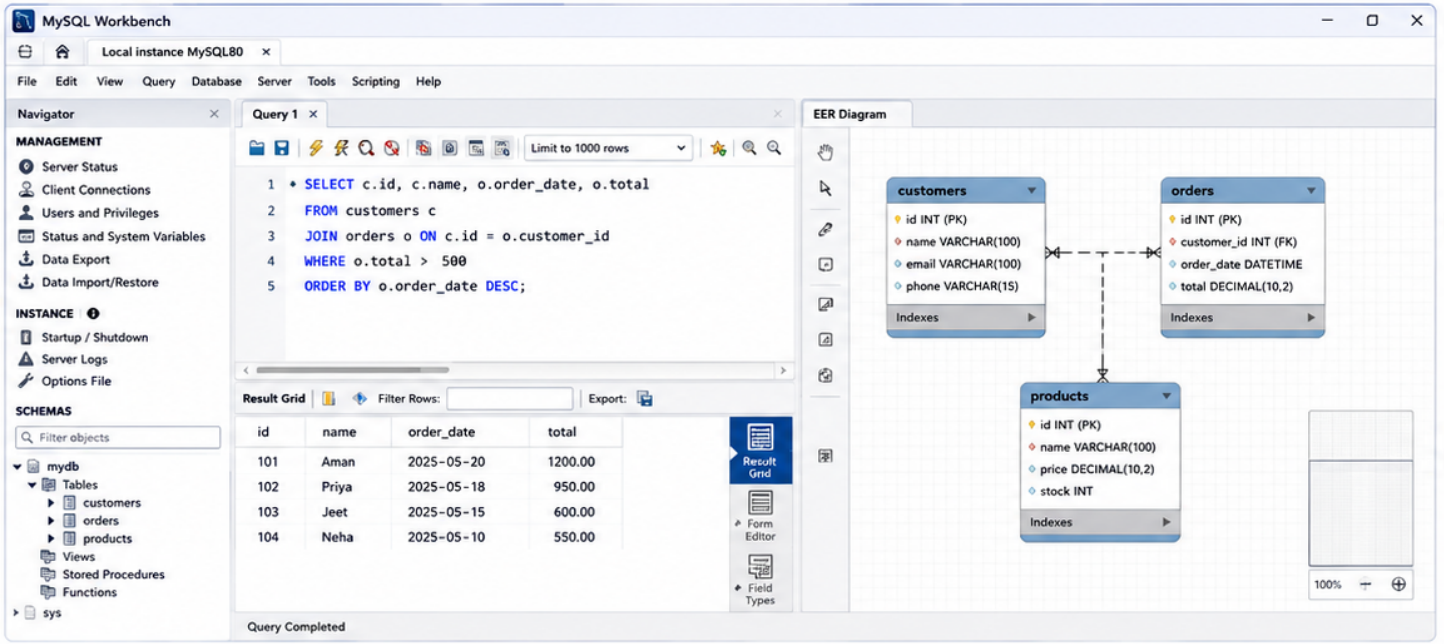
- SQL Development
- Database Design
- Data Modeling (ER Diagrams)
- Server Administration
- Backup, Import & Export
- User & Security Management

KEY FEATURES

- Intuitive SQL Editor
- Visual ER Diagram Designer
- Database Administration
- User & Privileges Management
- Data Import / Export
- Server Monitoring & Management

SUPPORTED PLATFORMS

Windows | Linux | macOS



SQL EDITOR

- Write and run SQL queries
- Syntax highlighting
- Auto-complete & IntelliSense
- Query history
- Results grid & export

DATA EXPORT

Export database, schema or table data to:

- SQL File (.sql)
- CSV File (.csv)
- Excel File (.xlsx)
- JSON File (.json)
- HTML, XML & more

DATA IMPORT

Import data from:

- SQL File (.sql)
- CSV File (.csv)
- Excel File (.xlsx)
- JSON, XML & more

Easy and safe data transfer.

1. SQL EDITOR & QUERY

- Powerful SQL editor for writing and executing queries.
- Supports multiple tabs, query history, bookmarks and explain plan.

```
SELECT *
FROM users
WHERE status = 'active';
```

2. ER DIAGRAM DESIGNER

- Design your database visually with tables, relationships and keys.
- Forward & Reverse Engineering supported.

3. USER & PRIVILEGES

- Create and manage database users.
- Set passwords and authentication.
- Grant or revoke privileges easily.

User	Host	Role	Privileges
root	localhost	Admin	ALL PRIVILEGES
jeet	localhost	Read/Write	SELECT, INSERT, UPDATE
guest	%	Read Only	SELECT

4. ADMINISTRATION

- Monitor server status and performance.
- Manage variables, logs, and options.
- Backup, restore and server maintenance tools.

5. DATA EXPORT & IMPORT

- Export and import data in multiple formats.
- Ideal for backup, migration and data sharing.

SQL | CSV | Excel | JSON

6. SERVER CONNECTIONS

- Connect to multiple MySQL servers.
- Manage connections easily.
- Save and organize connections.

BENEFITS

- Secure & Reliable
- Improve Productivity
- Easy Collaboration & Sharing
- Better Database Design
- Safe Backup & Recovery

QUICK SUMMARY

Develop • Design • Administer • Import/Export • Secure

MySQL Workbench – All-in-One Database Solution!

MySQL PERFORMANCE OPTIMIZATION

Write Better Queries. Use Indexes. Optimize Database. Get Maximum Speed.

SPEED — Faster Queries
PERFORMANCE — Better Efficiency
STABILITY — High Reliability
OPTIMIZATION — Smart Queries
MAXIMUM SPEED — Better User Experience

WHY OPTIMIZATION MATTERS?

- Faster query execution
- Less server load
- Better scalability
- Improved user experience
- Lower resource usage
- Cost effective

1. USE INDEXES

Indexes speed up search by reducing data scanning.

```
CREATE INDEX idx_name
ON users(name);
```

2. USE EXPLAIN

EXPLAIN shows how MySQL executes your query.

```
EXPLAIN SELECT *
FROM orders
WHERE user_id = 101;
```

3. OPTIMIZE QUERIES

Write efficient queries. Avoid unnecessary operations.

- Use specific columns
- Use WHERE clause
- Use LIMIT
- Avoid functions on indexed columns
- Join only needed tables

</>

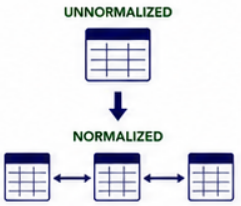
4. AVOID SELECT *

Select only required columns to reduce data transfer.

```
-- Avoid
SELECT * FROM users;
-- Better
SELECT id, name, email
FROM users;
```

5. NORMALIZATION

Normalize your database to reduce redundancy.



6. CACHING

Use caching to reuse frequently accessed data.

- Query Cache
- Application Cache (Redis)
- Object Cache
- Page Cache

BEST PRACTICES

- Use Indexes wisely
- Write optimized queries
- Avoid SELECT *
- Use Proper Data Types
- Normalize Database
- Use Caching
- Regularly Analyze & Optimize
- Monitor Performance

TOOLS FOR OPTIMIZATION

- EXPLAIN – Analyze query execution
- SHOW PROFILE – Detailed query profiling
- PERFORMANCE_SCHEMA – Detailed performance data
- SLOW QUERY LOG – Find slow queries
- OPTIMIZE TABLE – Optimize table storage

SQL EDITOR & EXPLAIN OUTPUT

SQL Editor

Limit to 1000 rows

```
1  SELECT o.id, o.order_date, u.name, o.total
2  FROM orders o
3  JOIN users u ON u.id = o.user_id
4  WHERE o.status = 'Completed'
5    AND o.order_date >= '2025-01-01'
   ORDER BY o.order_date DESC
   LIMIT 50;
```

EXPLAIN Result

id	select_type	table	type	possible_keys	key	rows	Extra
1	SIMPLE	u	eq_ref	PRIMARY	PRIMARY	1	Using where
1	SIMPLE	o	ref	idx_user_id,idx_status	idx_user_id	120	Using where; Using filesort

- **type:** eq_ref, ref → Good (uses indexes)
- **key:** idx_user_id → Index is used
- **rows:** 120 → Less rows scanned = Better Performance
- **Extra:** Using where, Using filesort → Can be optimized further

PERFORMANCE IMPROVEMENT AREAS

- **Reduce Full Table Scans** — Use indexes and proper filters.
- **Optimize Joins** — Join only necessary tables.
- **Filter Early** — Use WHERE clause early in queries.
- **Reduce Data Transfer** — Select only required columns.
- **Cache Frequently** — Store results and reuse.
- **Monitor & Tune** — Keep monitoring and improving.

EXAMPLE: BAD vs GOOD QUERY

BAD QUERY

```
SELECT * FROM orders;
```

→ Scans all columns and all rows

GOOD QUERY

```
SELECT id, order_date, total
FROM orders
WHERE status = 'Completed'
ORDER BY order_date DESC
LIMIT 50;
```

→ Uses index, returns fewer rows, faster

INDEX STRATEGY

Index Type	Use For	Example
PRIMARY KEY	Unique records	id (PK)
INDEX	Frequent searches	user_id, email
UNIQUE INDEX	Unique values	username
COMPOSITE INDEX	Multiple columns	(status, date)

Tip: Don't over-index. Only index what you query.

NORMALIZATION LEVELS

- 1NF Eliminate repeating groups
- 2NF Remove partial dependency
- 3NF Remove transitive dependency
- BCNF Stronger than 3NF

Normalization = Less redundancy, More consistency, Better performance

OPTIMIZE TABLE

Reclaim space and defragment tables.

```
OPTIMIZE TABLE orders;
```

QUERY PERFORMANCE TIPS

- ✓ Use LIMIT
- ✓ Use proper indexes
- ✓ Avoid functions in WHERE
- ✓ Use INNER JOIN by default
- ✓ Avoid OR in WHERE (use UNION)
- ✓ Keep statistics up to date
- ✓ Use appropriate data types

PERFORMANCE CHECKLIST

Indexes Used	Queries Optimized	Joins Optimized	Normalization Done	Caching Enabled	Monitoring Active

Optimize Today, Speed Up Always!

QUICK SUMMARY

Indexes · EXPLAIN · Optimize Query · Avoid SELECT * · Normalization · Caching · Performance · Speed

COMMON MySQL ERRORS

Understand Errors. Fix Faster. Write Better Queries.

DEBUGGING — Find & Fix Issues | ERROR LOG — Check Logs | BEST PRACTICES — Write Clean SQL | PERFORMANCE — Optimize Queries | RELIABLE — Stable Database

WHAT ARE MySQL ERRORS?

Errors are messages shown by MySQL when something goes wrong in your query, connection or database operation.

They help you identify the issue and fix it quickly.

QUICK TIPS

- Read error message carefully
- Check your query syntax
- Verify table & column names
- Check permissions
- Check data types
- Check server & connection
- Use logs for detailed info

1 SYNTAX ERROR

You have an error in your SQL syntax.

Error Message

```
You have an error in your SQL syntax;
check the manual that corresponds to your
MySQL server version for the right syntax
to use near 'SELEC * FROM users' at line 1
```

Example

```
SELEC * FROM users;      ✗ Wrong
SELECT * FROM users;     ✓ Correct
```

Check spelling, missing commas, quotes, parentheses and SQL keywords.

2 DUPLICATE ENTRY

Duplicate value for a unique key.

Error Message

```
ERROR 1062 (23000): Duplicate entry
'admin@example.com' for key 'email'
```

Example

```
INSERT INTO users (id, email)
VALUES (1, 'admin@example.com');
```

Ensure unique values for UNIQUE or PRIMARY KEY columns. Check existing records.

3 FOREIGN KEY ERROR

Cannot add or update a child row.

Error Message

```
ERROR 1452 (23000): Cannot add or update
a child row: a foreign key constraint fails
(`mydb`.`orders`, CONSTRAINT `fk_user`
FOREIGN KEY (`user_id`) REFERENCES `users`
(`id`))
```

Example

```
INSERT INTO orders (id, user_id)
VALUES (1, 99);  -- user_id 99 not exists
```

Make sure referenced parent record exists in the related table.

4 CONNECTION FAILED

Could not connect to MySQL server.

Error Message

```
ERROR 2003 (HY000): Can't connect to
MySQL server on 'localhost' (10061)
```

Example

```
mysql -u root -p
ERROR 2003 (HY000): Can't connect...
```

Check if MySQL server is running. Verify host, port, username and password.

5 UNKNOWN COLUMN

Unknown column in 'field list'.

Error Message

```
ERROR 1054 (42S22): Unknown column
'username' in 'field list'
```

Example

```
SELECT username FROM users;  -- column not exists
```

Check column name spelling. Use `DESCRIBE table_name;` to see correct column names.

6 PERMISSION DENIED

You don't have permission to do that.

Error Message

```
ERROR 1142 (42000): SELECT command
denied to user 'user'@'localhost' for
table 'users'
```

Example

```
SELECT * FROM users;  -- not allowed
```

Check user privileges using `SHOW GRANTS;` or contact your database administrator.

7 TIMEOUT ERROR

The connection or query timed out.

Error Message

```
ERROR 2013 (HY000): Lost connection to
MySQL server during query
```

Example

```
SELECT * FROM big_table;  -- takes too long
```

Optimize query, increase timeout values or check server performance.

8 DEBUGGING TIPS

Steps to find and fix the issue.

1. Read the full error message carefully.
2. Identify the error code (e.g., 1062, 1054).
3. Understand what the error means.
4. Check your query, data and server status.
5. Use logs and EXPLAIN to debug queries.
6. Test the fix with small steps.

CHIECK ERROR LOGS

Error logs provide more details about the problem.

```
-- Windows
SHOW VARIABLES LIKE 'log_error';

-- Linux
sudo cat /var/log/mysql/error.log
or
sudo cat /var/log/mysqld.log
```

USEFUL COMMANDS FOR DEBUGGING

Command	Purpose
`SHOW ERRORS;`	Show last error details
`SHOW WARNINGS;`	Show warnings
`SHOW GRANTS FOR 'user'@'host';`	Check user privileges
`DESCRIBE table_name;`	Show table structure
`EXPLAIN SELECT * FROM table;`	Analyze query performance
`SHOW PROCESSLIST;`	Show running queries and connections

ERROR CODES CATEGORY

Code Range	Meaning
1000 - 1999	Syntax, Access, Constraint Errors
2000 - 2999	Connection & Timeout Errors
3000 - 3999	Storage Engine Errors
4000 - 4999	Transaction Errors
5000 - 5999	Server Errors
6000+	Replication / Others

BEST PRACTICES TO AVOID ERRORS

- Write clean and well-structured SQL queries.
- Always validate data before inserting.
- Use proper indexes and constraints.
- Check permissions before running queries.
- Regularly backup your database.
- Monitor server performance and logs.

Read Errors Carefully | Understand the Cause | Apply the Right Fix | Test & Verify | Learn & Improve

DATABASE NORMALIZATION

Organize Data Efficiently • Reduce Redundancy • Improve Data Integrity

REDUCE REDUNDANCY | DATA INTEGRITY | BETTER PERFORMANCE | EASY MAINTENANCE | GOOD DESIGN

WHAT IS NORMALIZATION?

Normalization is the process of organizing data in a database to reduce redundancy and improve data integrity.

BENEFITS

- Eliminates data redundancy
- Prevents update anomalies
- Improves data integrity
- Saves storage space
- Easier data maintenance
- Scalability and consistency

NORMAL FORMS OVERVIEW

- 1NF First Normal Form
- 2NF Second Normal Form
- 3NF Third Normal Form
- BCNF Boyce-Codd Normal Form

KEY TERMS

- Candidate Key : Minimal set of attributes that uniquely identifies a row.
- Primary Key : Candidate key chosen to uniquely identify rows.
- Functional Dependency : One attribute is determined by another.

1NF FIRST NORMAL FORM

Eliminate repeating groups.
Ensure atomic (indivisible) values.

UNNORMALIZED TABLE

OrderID	Customer	Phones
101	Aman	9876543210, 9123456780
102	Priya	9988776655
103	Jeet	9871112222, 9333334444

↓

1NF TABLE

OrderID	Customer	Phone
101	Aman	9876543210
101	Aman	9123456780
102	Priya	9988776655
103	Jeet	9871112222
103	Jeet	9333334444

2NF SECOND NORMAL FORM

Remove partial dependency.
Every non-key attribute must fully depend on the whole primary key.

1NF TABLE (Has Partial Dependency)

OrderID	ProductID	ProductName	Quantity
101	P1	Laptop	1
101	P2	Mouse	2
102	P1	Laptop	1

↓

2NF TABLES

Orders

OrderID	Customer
101	Aman
102	Priya

OrderDetails

OrderID	ProductID	Quantity
101	P1	1
101	P2	2
102	P1	1

3NF THIRD NORMAL FORM

Remove transitive dependency.
Non-key attributes should not depend on other non-key attributes.

2NF TABLE (Has Transitive Dependency)

OrderID	CustomerID	CustomerName	City
101	C1	Aman	Delhi
102	C2	Priya	Mumbai

↓

3NF TABLES

Orders

OrderID	CustomerID
101	C1
102	C2

Customers

CustomerID	CustomerName	City
C1	Aman	Delhi
C2	Priya	Mumbai

BCNF BOYCE-CODD NORMAL FORM

For every functional dependency
X → Y, X must be a super key.

3NF TABLE (May have Anomaly)

StudentID	CourseID	InstructorID	InstructorName
S1	C1	I1	Raj
S1	C2	I2	Neha
S2	C1	I1	Raj

↓

BCNF TABLES

StudentCourse

StudentID	CourseID
S1	C1
S1	C2
S2	C1

CourseInstructor

CourseID	InstructorID
C1	I1
C2	I2

EXAMPLE: ORDER SYSTEM

UNNORMALIZED TABLE

OrderID	CustomerName	CustomerCity	ProductID	ProductName	Quantity
101	Aman	Delhi	P1	Laptop	1
101	Aman	Delhi	P2	Mouse	2
102	Priya	Mumbai	P1	Laptop	1

FUNCTIONAL DEPENDENCIES

- OrderID → CustomerName, CustomerCity
- ProductID → ProductName
- (OrderID, ProductID) → Quantity
- CustomerName → CustomerCity (Transitive)

NORMALIZATION JOURNEY

Unnormalized (Redundant Data) → 1NF (Atomic Values) → 2NF (Remove Partial Dependency) → 3NF (Remove Transitive Dependency) → BCNF (Strong Consistency)

NORMAL FORMS RULES

1NF	• All columns contain atomic values • No repeating identifes or arrays
2NF	• Must be in 1NF • No partial dependency on composite key
3NF	• Must be in 2NF • No transitive dependency
BCNF	• Must be in 3NF • Every determinant is a candidate key

COMMON ANOMALIES (SOLVED BY NORMALIZATION)

INSERT ANOMALY
Cannot insert data without other related data.

UPDATE ANOMALY
Updating one value requires multiple rows to change.

DELETE ANOMALY
Deleting a row causes unintended data loss.

BEST PRACTICES

- Always design tables up to 3NF or BCNF.
- Normalize without over-normalizing.
- Denormalize only for performance (carefully).
- Use proper keys and relationships.
- Test design with real data and scenarios.

QUICK CHECKLIST

- ✓ Remove Repeating Groups
- ✓ Eliminate Partial Dependencies
- ✓ Remove Transitive Dependencies
- ✓ Check Candidate Keys
- ✓ Validate with BCNF Rules

Better Design | Clean Data | Strong Integrity | High Performance | Scalable Database

DENORMALIZATION

Duplicate Data Strategically • Improve Read Performance • Speed Up Queries

SPEED — Faster Queries
PERFORMANCE — High Performance
READ OPTIMIZATION — Optimized Reads
DATA WAREHOUSE — Analytics Ready
DATABASE DESIGN — Better Design

WHAT IS DENORMALIZATION?

Denormalization is the process of intentionally adding redundant data or combining tables to improve read performance and reduce complex joins.

KEY BENEFITS

- Faster queries and reporting
- Reduces complex joins
- Better read performance
- Ideal for read-heavy systems
- Improves aggregation speed
- Useful in Data Warehouses

NORMALIZATION vs DENORMALIZATION

NORMALIZATION (Write Optimized)

- Minimal redundant data
- More tables and relationships
- Slower reads (more joins)
- Ideal for OLTP systems

DENORMALIZATION (Read Optimized)

- Contains redundant data
- Fewer tables and joins
- Faster reads and reports
- Ideal for Analytics / Reporting

EXAMPLE: E-COMMERCE SYSTEM

NORMALIZED DESIGN (More Tables)

orders

OrderID (PK)	CustomerID (FK)	OrderDate
TotalAmount		
101	1	2024-05-01
102	2	2024-05-02
103	1	2024-05-03

customers

CustomerID (PK)	CustomerName	Email	City
1	Aman	aman@email.com	Delhi
2	Priya	priya@email.com	Mumbai

order_items

OrderID (FK)	ProductID (PK)	ProductName	Quantity	Price
101	P1	Laptop	1	1000
101	P2	Mouse	2	250
102	P1	Laptop	1	2000

Query Example (Normalized)

```
SELECT o.OrderID, c.CustomerName, o.OrderDate, oi.ProductName, oi.Quantity, oi.Price
FROM orders o
JOIN customers c ON o.CustomerID = c.CustomerID
JOIN order_items oi ON o.OrderID = oi.OrderID;
```

More Joins
Complex Queries
Slower Read Performance

DENORMALIZED DESIGN (Fewer Tables)

orders_denorm

OrderID	OrderDate	CustomerName	CustomerEmail	CustomerCity	ProductName	Quantity	Price	TotalAmount
101	2024-05-01	Aman	aman@email.com	Delhi	Laptop	1	1000	1500
101	2024-05-01	Aman	aman@email.com	Delhi	Mouse	2	250	1500
102	2024-05-02	Priya	priya@email.com	Mumbai	Laptop	1	2000	2500
103	2024-05-03	Aman	aman@email.com	Delhi	Laptop	1	3200	3200

Query Example (Denormalized)

```
SELECT OrderID, CustomerName, OrderDate,
       ProductName, Quantity, Price, TotalAmount
FROM orders_denorm;
```

No Joins
Simple Queries
Faster Read Performance

HOW DENORMALIZATION IMPROVES PERFORMANCE

1. DUPLICATE DATA — Store related data together to avoid multiple table lookups and joins.

2. FAST QUERIES — Simpler queries mean database can return results much faster.

3. READ OPTIMIZATION — Optimized for SELECT operations. Ideal for reports and dashboards.

4. AGGREGATION SPEED — Pre-computed totals and summaries give instant aggregated results.

5. BETTER PERFORMANCE — Less CPU, less I/O, less memory usage, more speed.

6. DATA WAREHOUSE READY — Perfect for OLAP queries and business intelligence tools.

EXAMPLE: SALES SUMMARY (AGGREGATION)

Normalized (Slower)

```
SELECT c.City, SUM(o.TotalAmount) AS TotalSales
FROM orders o
JOIN customers c ON o.CustomerID = c.CustomerID
GROUP BY c.City;
```

Requires JOIN and GROUP BY
Slower for large datasets

Denormalized (Faster)

```
SELECT CustomerCity AS City, SUM(TotalAmount) AS TotalSales
FROM orders_denorm
GROUP BY CustomerCity;
```

No JOIN required
Faster aggregation

PERFORMANCE COMPARISON

Aspect	Normalized	Denormalized
Number of Tables	More	Fewer
Joins Required	More	Less/None
Read Performance	Slower	Faster
Write Performance	Faster	Slower
Storage Usage	Less	More
Maintenance	Easier	Harder
Best For	OLTP Systems	OLAP / Reporting

WHEN TO USE DENORMALIZATION?

- Read-heavy applications
- Complex reporting / analytics
- Data Warehouses
- Frequently used aggregations
- When performance is critical
- Caching or materialized views

POTENTIAL TRADE-OFFS

- Data redundancy
- More storage space
- Data inconsistency risk
- Complex write operations
- Harder to maintain
- Slower updates and deletes

DENORMALIZATION TECHNIQUES

- Adding redundant columns
- Combining tables
- Pre-calculated aggregates
- Materialized views
- Flattening relationships
- Data duplication

BEST PRACTICES

- Denormalize only when needed.
- Keep original normalized model as source of truth.
- Use ETL processes to sync denormalized data.
- Document all redundant data and dependencies.
- Monitor data consistency regularly.
- Test performance before and after denormalization.

DENORMALIZATION IN SHORT

Duplicate Data Strategically | Reduce Joins | Faster Queries | Read Optimized Design | Better Performance & Speed | Perfect for Analytics & Reporting | Powerful for Data Warehouse

ACID PROPERTIES

Ensuring Reliable Data in Database Transactions

TRANSACTIONS | RELIABLE DATA | INTEGRITY | DATABASE | TRUST & SAFETY

WHAT IS ACID?

ACID properties guarantee that database transactions are processed reliably, even in case of errors, crashes or power failures.

WHY ACID MATTERS?

- Prevents data loss
- Maintains data integrity
- Handles failures gracefully
- Ensures correct results
- Builds trust in your database

TRANSACTION EXAMPLE

Transfer ₹1000 from Account A to Account B

WITHOUT ACID (RISK)

- Partial updates (incomplete data)
- Data inconsistency
- Lost updates in concurrency
- Data loss on system failure
- Trust & business impact

A ATOMICITY

All operations in a transaction succeed completely, or none of them are applied.

Example
If money transfer fails in between, neither the amount is deducted from source nor added to destination.

All or Nothing

C CONSISTENCY

A transaction brings the database from one valid state to another. All rules and constraints are enforced.

Example
Data remains valid before and after the transaction. Constraints like Primary Key, Foreign Key, CHECK, UNIQUE are not violated.

Valid State to Valid State

I ISOLATION

Concurrent transactions do not interfere with each other. Each transaction is isolated until it is completed.

Example
If two users update data at the same time, isolation ensures one completes first, then the other, without conflicts.

Concurrency Without Conflicts

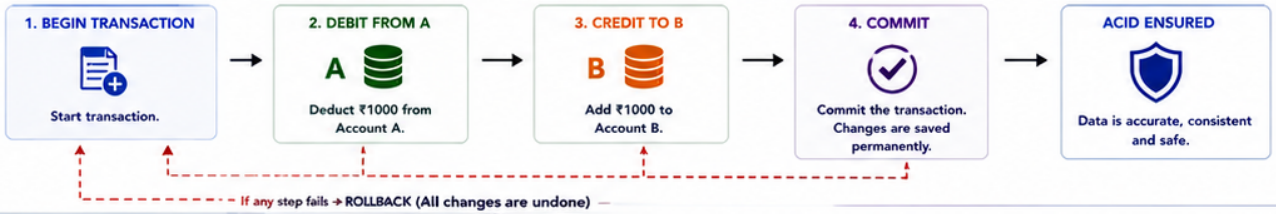
D DURABILITY

Once a transaction is committed, it remains permanent even in case of system failure, crash or power loss.

Example
After successful commit, data is saved permanently to disk. It will not be lost even if the system crashes.

Committed Data is Permanent

HOW ACID WORKS (MONEY TRANSFER EXAMPLE)



ACID SUMMARY

Property	Meaning	Focus
A Atomicity	All or Nothing	Reliability
C Consistency	Valid State to Valid State	Integrity
I Isolation	Concurrency Without Conflicts	Correctness
D Durability	Committed Data is Permanent	Persistence

ACID IN ACTION

1. **Begin Transaction**
   Start a new transaction.
2. **Perform Operations**
   Execute all required read/write operations.
3. **Validate & Check Rules**
   Ensure all constraints and rules are satisfied.
4. **Commit**
   Save changes permanently (ACID ensured).

BENEFITS

- Reliable & Safe Transactions
- Data Accuracy & Integrity
- Error Handling & Recovery
- High Data Consistency
- Better User Trust
- Enterprise Grade Applications

KEY TERMS

Transaction	:	A logical unit of work.
Commit	:	Make changes permanent.
Rollback	:	Undo changes on failure.
Concurrency	:	Multiple transactions at the same time.
Integrity	:	Accuracy and consistency of data.

ACID = Reliable Transactions | Reliable Data | Strong Integrity | Better Performance | Business Trust | Powerful Database

MySQL STORAGE ENGINES

Different Engines. Different Strengths. Choose the Best for Your Needs.

PERFORMANCE — High Speed
RELIABILITY — ACID & Safe
FLEXIBILITY — Multiple Options
EFFICIENCY — Optimize Storage
SCALABILITY — Grow with Ease
SECURITY — Data Protection

WHAT IS A STORAGE ENGINE?

A Storage Engine is a component that handles how data is stored, retrieved, updated and deleted in MySQL.

WHY STORAGE ENGINE MATTERS?

- Affects performance
- Impacts data integrity & security
- Determines features availability
- Improves scalability
- Helps optimize storage usage
- Choose the right engine for the right workload

CHECK CURRENT ENGINE

```
SHOW TABLE STATUS\G

-- or --
SELECT table_name, engine
FROM information_schema.tables
WHERE table_schema = 'your_db';
```

CHANGE STORAGE ENGINE

```
ALTER TABLE table_name
ENGINE = engine_name;
```

Examples:

```
ALTER TABLE mytable ENGINE = InnoDB;
ALTER TABLE mytable ENGINE = MyISAM;
```

MYSQL STORAGE ENGINES COMPARISON

FEATURES	InnoDB	MyISAM	MEMORY	CSV	ARCHIVE
Description	Default transactional engine with full ACID support.	Non-transactional engine with table-level locking.	Stores data in RAM. Very fast, non-durable.	Stores data in CSV files. Human readable.	Stores highly compressed data, read-mostly.
Transactions	Yes (ACID)	No	No	No	No
Locking Level	Row-level locking	Table-level locking	Table-level locking	Table-level locking	Row-level locking (Insert only)
Foreign Key Support	Yes	No	No	No	No
Performance (Read)	★★★★☆	★★★★☆	★★★★★	★★★☆☆	★★★☆☆
Performance (Write)	★★★★☆	★★★★☆	★★★★★	★☆☆☆☆	★★☆☆☆
Crash Recovery	Yes	No	No (Data lost)	No	No
Data Compression	No	No	No	No	Yes (High Compression)
Use Case	General purpose, OLTP applications	Read-heavy, full-text search, legacy applications	Temporary tables, caching, sessions	Data exchange, logs, reports	Logging, auditing, historical data
Supports Indexes	Yes (B-Tree)	Yes (B-Tree, Full-Text)	Yes (Hash, BTREE)	No	Yes (Limited)
Default	Yes (Since MySQL 5.5+)	Earlier default (Before 5.5)	No	No	No

ENGINE AT A GLANCE

InnoDB
- ACID Compliant
- Row-level Locking
- Reliable & Secure
- Best for OLTP Systems

RELIABLE & POWERFUL

MyISAM
- Fast Read Performance
- Table-level Locking
- No Crash Recovery
- Best for Read-Heavy Data

FAST & SIMPLE

MEMORY
- In-Memory Storage
- Ultra Fast
- Data Lost on Restart
- Best for Temp Tables

SUPER FAST (IN RAM)

CSV
- Human Readable
- Easy Import/Export
- No Index Support
- Best for Data Exchange

SIMPLE & PORTABLE

ARCHIVE
- High Compression
- Insert Only
- Very Space Efficient
- Best for Log/Archive Data

COMPRESS & STORE

HOW TO SEE AVAILABLE ENGINES

```
SHOW ENGINES;
```

Engine	Support	Comment
InnoDB	DEFAULT	Supports transactions...
MyISAM	YES	Non-transactional engine
MEMORY	YES	Hash based, stored in memory
CSV	YES	CSV storage engine
ARCHIVE	YES	Archive storage engine

SET DEFAULT STORAGE ENGINE

```
-- Check current default engine
SHOW VARIABLES LIKE 'default_storage_engine';

-- Set default engine (in my.cnf)
default-storage-engine = InnoDB
```

BEST PRACTICES

- Use InnoDB for most Applications
- Use MyISAM for read-heavy & full-text data
- Use MEMORY for temporary or session data
- Use CSV for data import/export
- Use ARCHIVE for write-once, read-few data
- Always backup important data

KEY TAKEAWAY

No single engine is best for all. Choose the right storage engine based on your application needs for optimal performance, reliability and efficiency.

EXAMPLE: CREATE TABLE WITH ENGINE

```
CREATE TABLE users (
 id INT PRIMARY KEY AUTO_INCREMENT,
 name VARCHAR(100),
 email VARCHAR(100)
) ENGINE = InnoDB;
```

EXAMPLE: CONVERT TABLE ENGINE

```
ALTER TABLE users
ENGINE = InnoDB;
```

PERFORMANCE TIPS

- Use InnoDB for reliability & data integrity
- Add proper indexes for faster queries
- Regularly optimize tables (OPTIMIZE TABLE table_name;)
- Monitor performance using SHOW STATUS;

QUICK SUMMARY

Engine	Transactions	Speed	Safety	Compression	Use Case
InnoDB	★★★★★	★★★★☆	★★★★★	★☆☆☆☆	General Purpose
MyISAM	★☆☆☆☆	★★★★☆	★☆☆☆☆	★☆☆☆☆	Read-Heavy
MEMORY	★☆☆☆☆	★★★★★	★☆☆☆☆	★☆☆☆☆	Temp / Cache
CSV	★☆☆☆☆	★★★☆☆	★☆☆☆☆	★☆☆☆☆	Import / Export
ARCHIVE	★☆☆☆☆	★★☆☆☆	★☆☆☆☆	★★★★★	Logs / Archive

PICK THE RIGHT ENGINE

Right Engine
Better Performance
Happy Database!

INNODB vs MYISAM

Choose the Right Engine for Your Database Needs

TRANSACTIONS · FOREIGN KEYS · LOCKING · PERFORMANCE · RELIABILITY · COMPARISON

ABOUT ENGINES

Storage Engines determine how data is stored, retrieved, and managed in MySQL.

InnoDB is the default engine in MySQL 5.5+ and supports transactions, while MyISAM is faster for read operations but lacks transaction support.

WHEN TO USE INNODB?

- When you need transaction safety
- When using FOREIGN KEY constraints
- When data integrity is critical
- For applications with frequent INSERT/UPDATE/DELETE
- For crash recovery and reliability

WHEN TO USE MYISAM?

- When you need faster read performance
- For read-heavy applications
- When full-text indexing is needed (supported earlier than InnoDB)
- For smaller tables without transactions
- When storage space is a concern

FEATURE	INNODB	MYISAM
Transactions	Yes — Supports ACID transactions	No — Does not support transactions
Foreign Keys	Yes — Supports FOREIGN KEY constraints	No — Does not support foreign keys
Locking	Row-Level Locking — Locks only the rows being modified	Table-Level Locking — Locks the entire table during write
Performance (Read)	★★★★☆ Good read performance	★★★★★ Excellent read performance
Performance (Write)	★★★☆☆ Slightly slower writes (due to logging)	★★★★☆ Faster write operations
Reliability	High — Crash safe, supports recovery	Lower — Not crash-safe, can corrupt
Storage	Uses more disk space — Due to indexes & logs	Uses less disk space — More efficient storage
Indexes	Supports B-Tree indexes	Supports B-Tree indexes — Also supports Full-Text indexes
AUTO ID Auto Increment	Better auto-increment handling — No gaps on rollback	Auto-increment can have gaps — If a row is deleted or rollback occurs
Default Engine	Default in MySQL 5.5+	Default in versions < 5.5
Use Case	OLTP, High Concurrency Apps — Banking, E-commerce, CMS, etc.	Read-Heavy Applications — Reporting, Logging, Analytics, etc.

KEY POINTS

- InnoDB ensures data integrity with transactions & foreign keys.
- MyISAM is faster for reads and uses less disk space.
- InnoDB is more reliable and crash-safe.
- Choose engine based on your application requirements.

QUICK SUMMARY

Factor	InnoDB	MyISAM
Transactions	Yes	No
Foreign Keys	Yes	No
Locking	Row-Level	Table-Level
Read Speed	Good	Excellent
Write Speed	Good	Faster
Reliability	High	Lower
Storage	More	Less
Best For	OLTP Apps	Read-Heavy Apps

OUR RECOMMENDATION

For most modern applications, InnoDB is the recommended choice!

EXAMPLE COMMANDS

```
-- Create table with InnoDB
CREATE TABLE users (
  id INT PRIMARY KEY,
  name VARCHAR(100)
) ENGINE = InnoDB;
```

```
-- Create table with MyISAM
CREATE TABLE users (
  id INT PRIMARY KEY,
  name VARCHAR(100)
) ENGINE = MyISAM;
```

PERFORMANCE OVERVIEW (EXAMPLE)

RELIABILITY COMPARISON

InnoDB	MyISAM
Crash Recovery	No Crash Recovery
ACID Compliance	No Transactions
Data Integrity	Can Corrupt
Safe Transactions	Less Data Integrity

BOTTOM LINE

Use InnoDB for reliability, transactions and data integrity.

Use MyISAM for speed and read-heavy workloads.

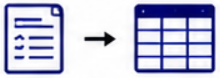
DATABASE VIEWS

Virtual Tables from Queries – Simplify, Secure & Reuse

VIRTUAL TABLE — Not Real Data
REPORTS — Easy Reporting
SECURITY — Restrict Access
REUSABLE QUERY — Save & Reuse
READ ONLY — Safe Views
PERFORMANCE — Optimize

WHAT IS A VIEW?

A View is a virtual table based on the result of a SELECT query. It does not store data physically.



BENEFITS OF VIEWS

- Simplify complex queries
- Improve security
- Provide data abstraction
- Create reusable queries
- Restrict access to sensitive data
- Support read-only reporting
- Help in logical data independence

KEY POINTS

- A view is a stored SELECT query.
- Views do not store data.
- Changes in base tables reflect in views.
- Some views can be updated (If they meet conditions).
- Views can be used like tables in SELECT statements.

VIEW COMMANDS

1. CREATE VIEW

Create a new view

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Example:

```
CREATE VIEW v_emp AS
SELECT emp_id, name, dept_id
FROM employees
WHERE status = 'Active';
```

2. ALTER VIEW

Modify an existing view

```
ALTER VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Example:

```
ALTER VIEW v_emp AS
SELECT emp_id, name, dept_id, salary
FROM employees
WHERE status = 'Active';
```

3. DROP VIEW

Delete a view

```
DROP VIEW view_name;
```

Example:

```
DROP VIEW v_emp;
```

4. USE VIEW

Query a view like a table

```
-- Use the view
SELECT * FROM view_name;
SELECT column1, column2
FROM view_name
WHERE condition;
```

Example:

```
SELECT * FROM v_emp;
SELECT name, salary FROM v_emp
WHERE salary > 50000;
```

EXAMPLE SCENARIO

Base Table: employees

emp_id	name	dept_id	salary	status
1	Aman	10	60000	Active
2	Priya	20	55000	Active
3	Raj	10	70000	Inactive
4	Neha	30	45000	Active

```
CREATE VIEW v_active_emp AS
SELECT emp_id, name, dept_id, salary
FROM employees
WHERE status = 'Active';
```

View: v_active_emp (Virtual Table)

emp_id	name	dept_id	salary
1	Aman	10	60000
2	Priya	20	55000
4	Neha	30	45000

USE CASES OF VIEWS

Reports — Create simplified views for reports and dashboards.

Security — Restrict access to sensitive columns by showing limited data.

Data Abstraction — Hide complex joins and logic from end users.

Reusable Query — Save frequently used queries as views.

Read-Only Access — Provide read-only data to users.

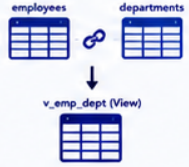
TYPES OF VIEWS

Simple View — Based on a single table.

Complex View — Based on multiple tables, joins, subqueries, aggregations.

CREATE VIEW WITH JOIN (EXAMPLE)

```
CREATE VIEW v_emp_dept AS
SELECT e.emp_id, e.name, d.dept_name, e.salary
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id
WHERE e.status = 'Active';
```



READ-ONLY VIEW EXAMPLE

```
CREATE VIEW v_high_salary AS
SELECT emp_id, name, salary
FROM employees
WHERE salary > 60000
WITH CHECK OPTION;
```

WITH CHECK OPTION ensures that INSERT/UPDATE through the view must satisfy the view condition.

CHARACTERISTICS OF VIEWS

- No physical data is stored.
- Reflects changes in base tables.
- May impact performance on complex views.
- Some views are updatable.
- Can be used in SELECT statements like tables.

QUICK SUMMARY

Command	Purpose	Example
CREATE VIEW	Create a new view	CREATE VIEW v AS SELECT ...
ALTER VIEW	Modify an existing view	ALTER VIEW v AS SELECT ...
DROP VIEW	Delete a view	DROP VIEW v;
SELECT FROM VIEW	Query data from view	SELECT * FROM v;

BEST PRACTICES

- Use views to simplify complex queries.
- Do not use views for heavy write operations.
- Use views to restrict access to sensitive data.
- Use meaningful names for views.
- Regularly review and optimize views.

Simplify Queries | Enhance Security | Reusable & Maintainable | Better Reporting | Reliable Data Access

MySQL EVENTS

Schedule Tasks • Automate Jobs • Save Time

SCHEDULER — Built-in Scheduler
AUTOMATION — Automate Tasks
RECURRING — Repeat Easily
DATABASE JOBS — Background Jobs
RELIABLE — Reliable Execution
TIMING — Precise Timing

WHAT ARE EVENTS?

Events are scheduled tasks in MySQL that run automatically at a specified time. They help in executing repetitive jobs without manual intervention.

MySQL Database → Event Scheduler → Task Executed

BENEFITS

- Automate repetitive tasks
- Run tasks at specific time
- Reduce manual work
- Improve efficiency
- Useful for reports, cleanups, backups, alerts, etc.

EVENT SCHEDULER STATUS

Check Status:
```
SHOW VARIABLES LIKE 'event_scheduler';
```

Enable Scheduler:
```
SET GLOBAL event_scheduler = ON;
```

Disable Scheduler:
```
SET GLOBAL event_scheduler = OFF;
```

EVENT COMMANDS

1 CREATE EVENT

Create a new scheduled event

```
CREATE EVENT event_name
ON SCHEDULE schedule
DO
BEGIN
  -- SQL statements
END;
```

Example:
```
CREATE EVENT daily_report
ON SCHEDULE EVERY 1 DAY
STARTS '2024-05-20 08:00:00'
DO
BEGIN
  INSERT INTO reports_log(message)
  VALUES('Daily report generated');
END;
```

2 ALTER EVENT

Modify an existing event

```
ALTER EVENT event_name
ON SCHEDULE new_schedule
DO
BEGIN
  -- New SQL statements
END;
```

Example:
```
ALTETE EVENT daily_report
ON SCHEDULE EVERY 1 DAY
STARTS '2024-05-20 09:00:00';
```

3 DROP EVENT

Delete an event

```
DROP EVENT [IF EXISTS] event_name;
```

Example:
```
DROP EVENT IF EXISTS daily_report;
```

4 SHOW EVENTS

View all events

```
SHOW EVENTS
[FROM db_name]
[LIKE 'pattern'];
```

Example:
```
SHOW EVENTS;
```

SCHEDULE OPTIONS

AT (One Time)

Run once at specific date & time

```
ON SCHEDULE AT '2024-05-25 10:30:00'
```

EVERY (Repeated Intervals)

Run repeatedly at fixed intervals

```
ON SCHEDULE EVERY 1 HOUR
```

DAILY TASK

Run every day at a specific time

```
ON SCHEDULE EVERY 1 DAY
STARTS '2024-05-20 08:00:00'
```

WEEKLY TASK

Run on specific days of the week

```
ON SCHEDULE EVERY 1 WEEK
STARTS '2024-05-20 08:00:00'
```

COMMON USE CASES

- **Daily Reports** — Generate daily sales or activity reports.
- **Data Cleanup** — Delete old logs or temporary data automatically.
- **Database Backup** — Automate backups at specific time.
- **Alerts & Notifications** — Send alerts or notifications based on conditions.
- **Data Sync** — Synchronize data between tables or systems.

EVENT TIMING OPTIONS

AT	Run once at a specific date & time
EVERY	Run repeatedly at intervals
STARTS	Start time of the first execution
ENDS	End time for the event (optional)
ON COMPLETION	NOT PRESERVE | PRESERVE

EXAMPLE: DAILY CLEANUP TASK

```
CREATE EVENT cleanup_logs
ON SCHEDULE EVERY 1 DAY
STARTS CURRENT_TIMESTAMP + INTERVAL 1 DAY
DO
BEGIN
 DELETE FROM app_logs
 WHERE log_date < CURRENT_DATE - INTERVAL 30 DAY;
END;
```

Deletes logs older than 30 days every day.

EXAMPLE: WEEKLY REPORT TASK

```
CREATE EVENT weekly_report
ON SCHEDULE EVERY 1 WEEK
STARTS '2024-05-20 09:00:00'
DO
BEGIN
 INSERT INTO reports_log(message)
 VALUES(CONCAT('Weekly report generated on ', NOW()));
END;
```

Generates weekly report every Monday 9 AM.

VIEW ALL EVENTS INFO

```
SHOW EVENTS\G
*************************** 1. row ***************************
Db                 : test_db
Name               : daily_report
Event              : CREATE DEFINER=`root`@`localhost` EVENT `daily_report`
Status             : ENABLED
On_schedule        : YES
Starts             : 2024-05-20 08:00:00
Ends               : NULL
Interval_value     : 1
Interval_field     : DAY
Time_zone          : SYSTEM
Created            : 2024-05-19 12:30:00
Last_altered       : 2024-05-19 12:30:00
Last_executed      : NULL
Next_execution     : 2024-05-20 08:00:00
*************************** 1. row ***************************
```

BEST PRACTICES

- Keep event tasks small and fast.
- Test events before enabling in production.
- Monitor event execution using SHOW EVENTS.
- Handle errors inside the event task.
- Use meaningful names for easy management.

Automate today, Save time every day!

MY SQL EVENTS IN SHORT

- CREATE EVENT — Schedule a task
- SCHEDULER — Runs automatically
- DAILY / WEEKLY TASKS — Repeat with ease
- AUTOMATION — Reduce manual work
- DATABASE JOBS — Run in background
- TIMING CONTROL — Precise execution
- SAVE TIME — Increase efficiency

SCHEDULED JOBS

Automate • Schedule • Maintain • Optimize

EVENT SCHEDULER — Built-in Scheduler
CRON JOBS — OS Level Scheduler
AUTOMATIC BACKUP — Protect Your Data
CLEANUP TASKS — Remove Old Data
AUTOMATION — Save Time & Effort
MAINTENANCE — Keep DB Healthy

WHAT ARE SCHEDULED JOBS?

Scheduled jobs are automated tasks that run at specific times to perform repetitive work without manual intervention.

BENEFITS

- Runs tasks automatically
- Saves time and manual effort
- Improves database performance
- Ensures regular maintenance
- Prevents data loss
- Keeps database clean & optimized
- Reliable and consistent execution

TYPES OF SCHEDULED JOBS

1. EVENT SCHEDULER (MySQL Built-in)

Use MySQL Event Scheduler to run SQL statements at specific time.

```
CREATE EVENT daily_report
ON SCHEDULE EVERY 1 DAY
STARTS '2024-05-25 01:00:00'
DO
  CALL generate_report();
```

Use Case:
Automate database tasks inside MySQL.

2. CRON JOBS (OS Level)

Use OS Cron Jobs to run scripts, commands or SQL queries.

```
0 2 * * * /usr/bin/mysql -u root
-pPassword database < /backup/
backup.sql
```

Use Case:
Run backups, scripts, or maintenance commands.

3. AUTOMATIC BACKUP

Schedule regular backups to protect your important data.

```
mysqldump -u root -pPassword \
database > /backup/db_$(date
+\%F).sql
```

Use Case:
Daily/Weekly backups to prevent data loss.

4. CLEANUP TASKS

Automate cleanup of old logs, temp data or unused records.

```
DELETE FROM temp_logs
WHERE created_at <
NOW() - INTERVAL 30 DAY;
```

Use Case:
Keep database small, fast & optimized.

5. MAINTENANCE TASKS

Schedule maintenance like OPTIMIZE, ANALYZE, or REPAIR.

```
OPTIMIZE TABLE large_table;
ANALYZE TABLE important_table;
```

Use Case:
Improve performance and fix table issues.

COMMON USE CASES

- **Daily Backup** — Backup database every day at a fixed time.
- **Old Data Cleanup** — Delete logs or old data regularly.
- **Monthly Reports** — Generate and email reports automatically.
- **Sync Data** — Sync data between databases or systems.
- **System Maintenance** — Optimize, analyze or repair tables automatically.

HOW IT WORKS

1. **CREATE JOB** — Create event or cron job
2. **SET SCHEDULE** — Define when it should run
3. **EXECUTE TASK** — Run SQL, Script or Command
4. **COMPLETE** — Task executed successfully
5. **REPEAT** — Runs again as per schedule

SAMPLE SCHEDULES

Task	Type	Schedule	Description
Daily Backup	Event / Cron	Every day at 02:00 AM	Backup database daily
Weekly Report	Event / Cron	Every Sunday at 08:00 AM	Generate weekly report
Cleanup Logs	Event	Every day at 01:00 AM	Delete logs older than 30 days
Optimize Tables	Event	Every Sunday at 03:00 AM	Optimize database tables
Data Sync	Cron	Every 15 minutes	Sync data to remote server

BEST PRACTICES

- Enable Event Scheduler on MySQL.
- Use meaningful job names.
- Log job execution for monitoring.
- Test jobs before enabling in production.
- Use least resource intensive time.
- Backup before running critical jobs.

ENABLE EVENT SCHEDULER

```
-- Check Status
SHOW VARIABLES LIKE 'event_scheduler';

-- Enable Event Scheduler
SET GLOBAL event_scheduler = ON;
-- Disable Event Scheduler
SET GLOBAL event_scheduler = OFF;
```

EXAMPLE: CREATE EVENT

```
CREATE EVENT cleanup_logs
ON SCHEDULE EVERY 1 DAY
STARTS '2024-05-25 01:00:00'
DO
  DELETE FROM logs
  WHERE created_at < NOW() - INTERVAL 30 DAY;
```

EXAMPLE: CRON JOB (LINUX)

```
# Run backup every day at 2 AM
0 2 * * * /usr/bin/mysqldump -u root -pPassword \
    database > /backup/db_$(date +\%F).sql
```

CHECK EVENT STATUS

```
-- Show all events
SHOW EVENTS;

-- Show event details
SHOW CREATE EVENT event_name\G
```

JOB MONITORING

```
-- Check event execution history
SELECT * FROM information_schema.EVENTS;
```

AUTOMATE TODAY, SAVE TIME FOREVER!

- Schedule Smart — Run on Time
- Automate Tasks — Reduce Manual Work
- Secure Data — Backup Regularly
- Clean & Optimize — Keep DB Healthy
- Monitor Jobs — Track & Review
- Reliable System — Better Performance

MySQL BEST PRACTICES

Build Faster • Run Smarter • Keep Data Safe

PERFORMANCE — Optimize Queries
RELIABILITY — Protect Data
SECURITY — Secure Access
MAINTAINABILITY — Easy to Maintain
SCALABILITY — Grow with Confidance
QUALITY — Better Results

1 MEANINGFUL NAMES

Use clear and meaningful names for databases, tables, columns, and indexes.

- Use descriptive table names
- Use singular for table names
- Use consistent naming style
- Avoid abbreviations
- Be specific and self-explanatory

Example

```
Good: customer_orders
Bad : cust_ord
```

2 INDEXES

Use indexes to speed up queries, but avoid over-indexing.

- Index columns used in WHERE, JOIN, ORDER BY
- Avoid indexes on low cardinality columns
- Remove unused indexes
- Use composite indexes wisely

Example

```
CREATE INDEX idx_user_email
ON users(email);
```

3 NORMALIZATION

Normalize tables to reduce redundancy and improve integrity.

- Follow 1NF, 2NF, 3NF
- Reduce data duplication
- Use foreign keys
- Improve data consistency

Example

Split user and address into separate tables.

4 BACKUP

Always backup your database regularly.

- Take regular automated backups
- Test restore process
- Keep backups in a safe location
- Follow 3-2-1 backup rule (3 copies, 2 media, 1 offsite)

Example

```
mysqldump -u root -p db_name >
backup.sql
```

5 PARAMETERIZED QUERIES

Use parameterized queries to prevent SQL injection.

- Never trust user input
- Use prepared statements
- Validate and sanitize inputs
- Escape dynamic values

Example (Prepared Statement)

```
SELECT * FROM users
WHERE id = ?;
```

6 SECURITY

Secure your MySQL server and data.

- Use strong passwords
- Limit user privileges
- Use SSL connections
- Keep MySQL updated
- Disable unnecessary services

Example

```
CREATE USER 'app_user'@'localhost'
IDENTIFIED BY 'StrongPass123!';
GRANT SELECT, INSERT ON db.*
TO 'app_user'@'localhost';
```

7 DOCUMENTATION

Document your database structnize and logic.

- Document tables and columns
- Explain relationships
- Document stored procedures
- Optimize JOErnus updated
- Maintain change history

Example

```
EXPLAIN SELECT * FROM orders
 WHERE order_date > '2024-01-01';
```

QUICK TIPS

- Use LIMIT to restrict results
- Use appropriate data types
- Avoid NULLs if not needed
- Keep your database clean
- Monitor performance regularly
- Use transactions where needed

DOs AND DON'Ts

DOs

- Do use meaningful names
- Do create necessary indexes
- Do backup your data regularly
- Do validate user inputs
- Do keep your server updated
- Do test your queries

DON'Ts

- Don't use SELECT *
- Don't over-normalize
- Don't ignore backups
- Don't hardcode credentials
- Don't give erver privileges
- Don't ignore slow queries

EXAMPLE: BAD vs GOOD QUERY

BAD QUERY ☹

```
SELECT * FROM orders
WHERE status = 'Shipped'
ORDER BY created_at;
```

- Uses SELECT *
- May return large data
- No LIMIT
- Can be slow

→

GOOD QUERY ☺

```
SELECT id, customer_id, total,
created_at FROM orders
WHERE status = 'Shipped'
ORDER BY created_at DESC
LIMIT 100;
```

- Selects only needed columns
- Returns limited results
- Better performance

PERFORMANCE CHECKLIST

- Use Indexes
- Optimize Queries
- Avoid N+1 Queries
- Use Appropriate Data Types
- Monitor & Analyze

USEFUL COMMANDS

```
-- Show databases
SHOW DATABASES;

-- Show tables
SHOW TABLES;

-- Describe table
DESCRIBE table_name;

-- Explain query
EXPLAIN SELECT * FROM users;

-- Show indexes
SHOW INDEX FROM table_name;
```

REMEMBER

Good practices today
Better performance tomorrow!

- Clean Design — Easy to Maintain
- Fast Queries — Better Performance
- Secure Data — Safe & Reliable
- Regular Backup — Never Lose Data
- Document Everything — Know Your Database
- Optimize Continuously — Always Improve

MySQL SECURITY

Protect Your Data. Secure Your Database. Build Trust.

PROTECT Your Data | CONTROL Access | PREVENT Threats | ENSURE Compliance | BUILD Confidence

1 STRONG PASSWORD

Use strong, unique passwords for all users and applications to prevent unauthorized access.

Best Practices

- ✓ At least 12 characters
- ✓ Use letters, numbers, symbols
- ✓ Avoid common words
- ✓ Regularly change passwords

2 SSL

Encrypt data in transit using SSL/TLS to protect against eavesdropping and attacks.

Best Practices

- ✓ Enable SSL/TLS connections
- ✓ Use trusted certificates
- ✓ Enforce encryption
- ✓ Disable unencrypted protocols

3 USER ROLES

Create roles to manage access and permissions efficiently across your database.

Best Practices

- ✓ Use the principle of least privilege
- ✓ Group users by responsibilities
- ✓ Review roles regularly
- ✓ Avoid excessive privileges

4 GRANT

Grant only the permissions needed for users to perform their tasks.

Best Practices

- ✓ Grant specific privileges
- ✓ Limit access to necessary objects
- ✓ Use least privilege
- ✓ Review grants periodically

5 REVOKE

Revoke unnecessary privileges to minimize potential security risks.

Best Practices

- ✓ Revoke unused permissions
- ✓ Review user access regularly
- ✓ Remove unnecessary roles
- ✓ Audit privilege changes

6 LEAST PRIVILEGE

Give users only the minimum permissions necessary to perform their job.

Best Practices

- ✓ Follow the principle of least privilege
- ✓ Limit access to sensitive data
- ✓ Separate duties
- ✓ Regularly review permissions

7 FIREWALL

Use firewalls to restrict access to your MySQL server and block unauthorized connections.

Best Practices

- ✓ Allow only trusted IP addresses
- ✓ Block unnecessary ports
- ✓ Monitor firewall logs
- ✓ Keep firewall rules updated

8 SECURE CONNECTION

Use secure connection methods to protect data between your applications and MySQL.

Best Practices

- ✓ Use encrypted connections
- ✓ Avoid public networks
- ✓ Validate server certificates
- ✓ Use secure configuration

ADDITIONAL SECURITY TIPS

- Keep MySQL software up to date
- Disable unused features and services
- Enable auditing and logging
- Regularly backup your database
- Monitor for suspicious activity

SECURE BY DEFAULT

MySQL provides multiple security features—use them together for a layered defense and strong data protection.

MONITOR & AUDIT

Continuously monitor your database and audit access to detect and respond to threats quickly.

STAY PROTECTED

Security is an ongoing process. Review, update, and adapt to stay ahead of threats.

QUICK COMMANDS

```
> SHOW GRANTS FOR 'user'@'host';
> SHOW VARIABLES LIKE 'ssl%';;
> SELECT USER, HOST FROM mysql.user;
> CHECK TABLE mysql.user;
```

STRONG SECURITY STRONGER DATABASE | PROTECT TODAY SECURE TOMORROW | SAFE DATA BETTER FUTURE | SECURE MYSQL STRONGER SYSTEM | PEACE OF MIND EVERY CONNECTION

MySQL INTERVIEW QUESTIONS

Top Questions & Answers for Beginners to Advanced Level

BOOST YOUR PREP | CRACK INTERVIEWS | IMPROVE SKILLS | BE CONFIDENT

1 JOINS

- What is JOIN in MySQL?
- What are the types of JOINs?
- Difference between INNER JOIN and LEFT JOIN?
- Write a query to find employees and their departments.

Example (INNER JOIN)

```
SELECT e.name, d.dept_name
FROM employees e
INNER JOIN departments d
ON e.dept_id = d.dept_id;
```

2 INDEXES

- What is an Index in MySQL?
- How do indexes improve performance?
- What are the types of Indexes?
- How to create and drop an index?

Example (CREATE INDEX)

```
CREATE INDEX idx_name
ON employees(name);
```

3 PRIMARY KEY

- What is a Primary Key?
- Can a table have multiple Primary Keys?
- Can Primary Key contain NULL values?
- How Primary Key helps in a table?

Example

```
CREATE TABLE users (
  id INT PRIMARY KEY,
  name VARCHAR(50)
);
```

4 FOREIGN KEY

- What is a Foreign Key?
- How does Foreign Key maintain integrity?
- Can a table have multiple Foreign Keys?
- Write a query with Foreign Key.

Example

```
CREATE TABLE orders (
  order_id INT PRIMARY KEY,
  user_id INT,
  FOREIGN KEY (user_id) REFERENCES users(id)
);
```

5 NORMALIZATION

- What is Normalization?
- What are the Normal Forms (1NF, 2NF, 3NF, BCNF)?
- Why do we need Normalization?
- Explain 3NF with example.

Example (3NF)

Eliminate transitive dependency and ensure data is stored logically in separate tables.

6 TRANSACTIONS

- What is a Transaction?
- What are the ACID properties?
- What are COMMIT, ROLLBACK, SAVEPOINT?
- How do you handle transactions in MySQL?

Example

```
START TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT;
```

7 VIEWS

- What is a View in MySQL?
- How do you create, alter and drop a view?
- What are the advantages of Views?
- Are Views stored physically?

Example (CREATE VIEW)

```
CREATE VIEW emp_view AS
SELECT id, name, dept_id
FROM employees
WHERE status = 'Active';
```

8 TRIGGERS

- What is a Trigger in MySQL?
- What are the types of Triggers?
- When are Triggers executed?
- Write a sample trigger.

Example (BEFORE INSERT Trigger)

```
CREATE TRIGGER before_emp_insert
BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
   SET NEW.created_at = NOW();
END;
```

BONUS QUESTIONS

- [x] What is the difference between DELETE and TRUNCATE?
- [x] What is the difference between CHAR and VARCHAR?
- [x] What is the use of GROUP BY and HAVING?
- [x] What are Stored Procedures?
- [x] What is the difference between UNION and UNION ALL?
- [x] How does MySQL handle concurrency?

TIPS TO CRACK INTERVIEWS

- **Strong Basics** — Clear your fundamentals
- **Practice Queries** — Write and optimize queries daily
- **Understand Concepts** — Focus on why and how
- **Mock Interviews** — Practice & get feedback

USEFUL MySQL TOOLS

Manage • Develop • Optimize • Monitor

Easy Management | Better Performance | Secure & Reliable | Increase Productivity

1 MySQL Workbench

Official visual tool from Oracle for database design, development and administration.

Key Features

- Data modeling & ER diagrams
- SQL development & execution
- Server administration
- Performance monitoring

2 phpMyAdmin

Popular open-source web interface for MySQL and MariaDB.

Key Features

- Web-based database management
- Easy import/export of databases
- User management
- Run SQL queries online

3 DBeaver

Universal database tool supporting MySQL and many other databases.

Key Features

- Multi-database support
- Advanced SQL editor
- Data export in multiple formats
- ER diagrams & data visualization

4 HeidiSQL

Lightweight and fast MySQL client for Windows.

Key Features

- Easy to use interface
- Import/export data
- Query tab with syntax highlighting
- Server & session manager

5 Navicat

Powerful GUI tool for database administration, development and management.

Key Features

- Visual query builder
- Data synchronisration
- Backup & restore
- Support for multiple databases

6 DataGrip

Advanced MySQL IDE from JetBrains for developers.

Key Features

- Smart code completion
- Refactoring & navigation
- Database tools integration
- Version control integration

7 TablePlus

Modern, native GUI tool for MySQL with beautiful interface.

Key Features

- Intuitive and modern UI
- Query editor with autocomplete
- Multiple connections
- Table, row & column editing

8 MySQL CLI

Command Line Interface for MySQL – powerful and lightweight.

Key Features

- Fast and efficient
- Script automation
- Remote database access
- Perfect for developers & DevOps

Why Use These Tools?

- Save Time & Effort
- Improve Productivity
- Optimize Database Work
- Secure & Reliable Management

TOOLS AT A GLANCE

Tool	Platform	Interface	Best For	License	Price
1 MySQL Workbench	Windows, macOS, Linux	Desktop	DBA, Developers	GPL	Free
2 phpMyAdmin	Web (Any OS)	Web	Beginners, DB Admin	GPL	Free
3 DBeaver	Windows, macOS, Linux	Desktop	Developers, Analysts	Apache 2.0	Free
4 HeidiSQL	Windows	Desktop	DBA, Developers	GPL	Free
5 Navicat	Windows, macOS, Linux	Desktop	Professionals, Teams	Proprietary	Paid
6 DataGrip	Windows, macOS, Linux	Desktop	Developers	Proprietary	Paid
7 TablePlus	Windows, macOS, Linux	Desktop	Developers, Designers	Proprietary	Paid
8 MySQL CLI	Windows, macOS, Linux	Command Line	Developers, DevOps	GPL	Free

PRO TIP

- Use GUI tools for development and management.
- Use CLI for automation and scripting tasks.
- Always backup your database before making changes.
- Choose the tool that fits your workflow.

DATABASE ARCHITECTURE

From Client Request to Results

PERFORMANCE — Fast & Efficient
RELIABILITY — Secure & Stable
SCALABILITY — Grow Effortlessly
FLEXIBILITY — Highly Customizable

1 CLIENT

- Web Browser
- Mobile App
- Desktop App
- Other Clients

Request →
← Results

2 APPLICATION

- Business Logic
- Query Builder / ORM
- Connection Handling
- Send SQL Queries

SQL Query →

3 MySQL SERVER

- **Connection Management** — Handles client connections
- **Query Parser** — Parses and validates SQL
- **Query Optimizer** — Creates efficient execution plan
- **Execution Engine** — Executes the query
- **Cache & Buffer** — Improves performance

Read / Write →

4 DATABASE

- **Schemas / Databases** — Logical container
- **Tables** — Data organized in rows
- **Relationships** — Links between tables
- **Constraints** — Rules for data integrity

5 TABLES

- **Rows (Records)** — Individual data entries
- **Columns (Fields)** — Define data properties
- **Data Types** — Int, Varcher, Date, etc.
- **Constraints** — PK, FK, UNIQUE, NOT NULL

6 INDEXES

- **Speeds up search** — Faster data retrieval
- **Types of Indexes** — PRIMARY, UNIQUE, INDEX
- **Improves Performance** — Reduces query time
- **Uses Extra Storage** — For quick lookups

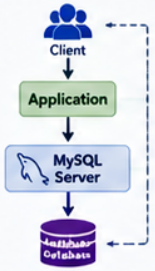
7 STORAGE ENGINE

- **InnoDB (Default)** — Transactions, ACID, FK
- **MyISAM** — Fast reads, No FK
- **Memory** — Int, Varmory Tables
- **Others** — Archive, CSV, Merge, etc.

RESULTS — Returned to Client

ARCHITECTURE FLOW

1. Client sends request to the application.
2. Application builds and sends SQL query.
3. MySQL Server processes the query.
4. Data is read from / written to the database.
5. Data is stored in tables (rows & columns).
6. Indexes help in faster data retrieval.
7. Storage Engine manages the physical data.



STORAGE ENGINE LAYER

Handles how data is stored, retrieved and managed on disk.

InnoDB
- ACID Compliant
- Transactions
- Foreign Keys
- Row-Level Locking
- Crash Recovery

MyISAM
- High Performance
- Table-Level Locking
- No Transactions
- No Foreign Keys
- Full Text Search

Memory
- In-Memory Tables
- Very Fast
- Data Lost on Restart
- For Temporary Data
- No Disk I/O

Others
- Archive Engine
- CSV Engine
- Merge Engine
- Federated Engine
- Custom Engines

KEY BENEFITS

- **High Performance** — Optimized query execution
- **Data Integrity** — Constraints & relationships ensure accuracy
- **Scalability** — Handles small to large applications
- **Reliability & Security** — Secure connections and data protection

QUERY EXECUTION FLOW

From SQL Query to Output

FAST · OPTIMIZED · SECURE · EFFICIENT

1 SQL QUERY

Client sends an SQL query to the MySQL Server.

Example

```
SELECT * FROM
users WHERE age > 20;
```

2 PARSER

Parser checks the syntax and parses the query into an internal structure.

Syntax Check

Parse Tree Generated

3 OPTIMIZER

Optimizer analyzes possible execution paths and chooses the best plan.

Best Plan Chosen

Based on cost, indexes, statistics

4 EXECUTION PLAN

Optimizer creates an execution plan with operations in the best order.

Plan Includes

- Table Access
- Join Order
- Index Usage
- Filters

5 STORAGE ENGINE

The Storage Engine executes the plan and accesses the required data.

Engines

- InnoDB (Default)
- MyISAM
- Memory
- Others

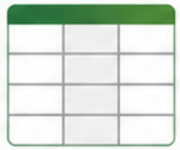
6 FETCH DATA

Rows are read, filtered and processed as per the execution plan.

Filter & Process

Apply WHERE, JOIN, GROUP BY, ORDER BY, etc.

7 RESULT SET



Processed rows are organized into the final result set.

Result Set

Ready to be returned to the client

8 OUTPUT

Result set is sent back to the client as output.

Send Back

Results displayed to the user

KEY POINTS

- MySQL processes queries in a structured and optimized way.
- Optimizer plays a key role in performance.
- Proper indexing and statistics help in better execution plans.
- Different storage engines handle data based on their design.

FLOW OVERVIEW

1 SQL Query → 2 Parser → 3 Optimizer → 4 Execution Plan → 5 Storage Engine → 6 Fetch Data → 7 Result Set → 8 Output

Optimized Execution = Better Performance

BENEFITS

- **High Performance** — Optimized queries run faster
- **Efficient Resource Usage** — Better use of CPU, Memory & Disk
- **Scalable** — Handles more users and large data
- **Reliable Results** — Accurate and consistent output

MySQL DEVELOPMENT WORKFLOW

From Idea to Deployment

STRUCTURED PROCESS | BETTER PERFORMANCE | FEWER ERRORS | SECURE & RELIABLE

1 DESIGN DATABASE

- Understand requirements
- Define entities
- Create ER Diagram
- Define relationships

OUTPUT
Database Design (ER Diagram)

2 CREATE TABLES

- Write CREATE TABLE statements
- Define columns, data types
- Set Primary & Foreign Keys
- Add Constraints

OUTPUT
Tables Created in MySQL

3 INSERT DATA

- Insert initial data
- Validate data integrity
- Use INSERT statements
- Use bulk insert if needed

OUTPUT
Data Inserted Successfully

4 WRITE QUERIES

```
SELECT *
FROM users
WHERE status = 'Active';
```

- Write SELECT, JOIN, UPDATE, DELETE etc.
- Use proper filters and conditions
- Follow best practices

OUTPUT
Queries Ready for Use

5 TEST

- Test queries & logic
- Check for errors
- Validate results
- Perform unit testing

OUTPUT
Tested & Validated

6 OPTIMIZE

- Analyze slow queries
- Add Indexes
- Optimize table structure
- Improve performance

OUTPUT
Optimized Performance

7 BACKUP

- Create regular backups
- Use mysqldump / Backup tools
- Store backups securely
- Test restore process

OUTPUT
Backup Stored Safely

8 DEPLOY

- Deploy to staging environment
- Final testing
- Deploy to production
- Monitor & maintain

OUTPUT
Application Deployed

WORKFLOW OVERVIEW

BEST PRACTICES

- Follow naming conventions
- Use proper data types
- Normalize your database
- Index important columns
- Validate all inputs
- Optimize queries regularly
- Backup regularly
- Monitor performance

TOOLS YOU CAN USE

MySQL Workbench | phpMyAdmin | DBeaver | HeidiSQL

TablePlus | DataGrip | mysqldump | Git

A structured workflow leads to better databases, high performance, and successful applications.

MySQL PROJECT STRUCTURE

Organize • Manage • Secure • Scale •

SECURE & RELIABLE | PERFORMANT & OPTIMIZED | WELL ORGANIZED | SCALABLE & MAINTAINABLE

1 DATABASE

The main container that holds all database objects and data.

Example

```
CREATE DATABASE myproject;
USE myproject;
```

2 TABLES

Tables store data in rows and columns. Each table has a specific purpose.

Example

- users
- products
- orders
- order_items

3 RELATIONSHIPS

Define relationships between tables using Primary Keys and Foreign Keys.

Example

```
orders.user_id → users.id
order_items.order_id → orders.id
```

4 VIEWS

Views are virtual tables based on query results. They simplify complex queries and improve security.

Example

```
CREATE VIEW v_orders AS
SELECT o.id, u.name, o.total
FROM orders o
JOIN users u ON o.user_id = u.id;
```

5 PROCEDURES

Stored procedures contain SQL logic that can be reused in the application.

Example

```
CREATE PROCEDURE GetUserOrders
(IN p_user_id INT)
BEGIN
  SELECT * FROM orders
  WHERE user_id = p_user_id;
END;
```

6 TRIGGERS

Triggers automatically execute actions before or after INSERT, UPDATE, DELETE operations.

Example

```
CREATE TRIGGER trg_user_insert
AFTER INSERT ON users
FOR EACH ROW
BEGIN
  INSERT INTO user_logs
  SET action = 'INSERT';
END;
```

7 USERS

Users access the database with defined roles and permissions for security and control.

Example

```
CREATE USER 'app_user'@'%'
IDENTIFIED BY 'StrongPass123!';
GRANT SELECT, INSERT, UPDATE
ON myproject.* TO 'app_user'@'%';
FLUSH PRIVILEGES;
```

8 BACKUPS

Backups protect your data and help in case of failure or data loss.

Example

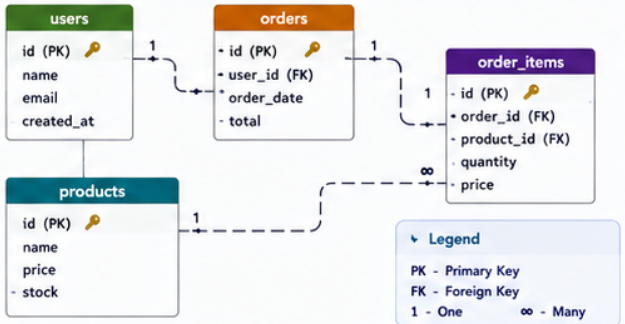
```
mysqldump -u root -p
myproject >
myproject_backup.sql
```

PROJECT STRUCTURE OVERVIEW

Benefits

- Well Organized
- Easy Maintenance
- High Performance
- Better Security
- Scalable Design
- Disaster Recovery

SAMPLE DATABASE STRUCTURE



BEST PRACTICES

- Use meaningful database and table names
- Define Primary Keys for each table
- Use Foreign Keys to maintain relationships
- Use Views for security and simplicity
- Use Triggers and Procedures for automation
- Manage users and permissions carefully
- Take regular backups of your database

Tools

- MySQL Workbench
- phpMyAdmin
- DBeaver
- HeidiSQL
- mysqldump

COMMON MySQL COMMANDS

Essential Commands Every MySQL User Should Know

Easy to Use | Fast Execution | Powerful | Beginner Friendly

1 SHOW DATABASES

Displays a list of all databases available on the MySQL server.

Syntax

```
SHOW DATABASES;
```

Example

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| myproject          |
| test               |
+--------------------+
```

2 SHOW TABLES

Displays a list of all tables in the current database.

Syntax

```
SHOW TABLES;
```

Example

```
mysql> SHOW TABLES;
+---------------------+
| Tables_in_myproject |
+---------------------+
| customers           |
| orders              |
| products            |
+---------------------+
```

3 DESCRIBE

Shows the structure of a table including column names, types and more.

Syntax

```
DESCRIBE table_name;
```

Example

```
mysql> DESCRIBE customers;
+------------+--------------+------+-----+---------+----------------+
| Field      | Type         | Null | Key | Default | Extra          |
+------------+--------------+------+-----+---------+----------------+
| id         | int          | NO   | PRI | NULL    | auto_increment |
| name       | varchar(100) | NO   |     | NULL    |                |
| email      | varchar(100) | YES  |     | NULL    |                |
| created_at | datetime     | YES  |     | NULL    |                |
+------------+--------------+------+-----+---------+----------------+
```

4 USE

Selects a database to work with.

Syntax

```
USE database_name;
```

Example

```
mysql> USE myproject;
Database changed
```

5 SOURCE

Runs SQL statements from a file.

Syntax

```
SOURCE file_path;
```

Example

```
mysql> SOURCE /path/to/script.sql;
Query OK, 0 rows affected (0.00 sec)
```

6 EXIT

Exits the MySQL command-line client.

Syntax

```
EXIT;
```

Example

```
mysql> EXIT;
Bye
```

7 STATUS

Displays server status information.

Syntax

```
STATUS;
```

Example

```
mysql> STATUS;
-------------
Uptime: 12345    Threads: 2    Questions: 567
Slow queries: 0  Opens: 15     Flush tables: 1
-------------
```

8 HELP

Displays help for MySQL commands.

Syntax

```
HELP command;
```

Example

```
mysql> HELP SELECT;
Name: 'SELECT'
Description:
Syntax: SELECT [DISTINCT] ...
```

Tip: You can also use \\h instead of HELP and \\s instead of STATUS in the MySQL CLI.

MySQL PROJECT CHECKLIST

Complete Checklist for a Strong, Secure & Performant MySQL Project

RELIABLE — Build with Confidence
PERFORMANT — Optimize for Speed
SECURE — Protect your Data
SUCCESSFUL — Deliver Quality

1 PRIMARY KEYS

- Define PRIMARY KEY for every table.
- Use auto-increment for unique ID.
- Avoid composite primary keys unless necessary.
- Ensure primary key is NOT NULL and UNIQUE.

Quick Tip: Primary Keys uniquely identify each record in a table.

2 FOREIGN KEYS

- Define FOREIGN KEY relationships.
- Ensure data integrity with valid references.
- Use ON DELETE and ON UPDATE rules.
- Avoid orphan records.

Quick Tip: Foreign Keys maintain relationships and integrity between tables.

3 INDEXES

- Index columns used in WHERE, JOIN, ORDER BY.
- Avoid over-indexing.
- Use appropriate index types (BTREE, FULLTEXT).
- Review and optimize indexes regularly.

Quick Tip: Indexes improve query performance but can slow down writes.

4 BACKUP

- Create regular backups.
- Test backup and restore process.
- Store backups in secure location.
- Have a disaster recovery plan.

Quick Tip: A good backup today saves you from big data loss tomorrow.

5 SECURITY

- Use strong passwords.
- Create users with least privilege.
- Enable SSL connections.
- Keep MySQL updated.

Quick Tip: Security is not optional, it's essential.

6 OPTIMIZATION

- Optimize slow queries.
- Use EXPLAIN to analyze queries.
- Normalize database design.
- Monitor and tune server configuration.

Quick Tip: Optimize early, scale easily.

7 TESTING

- Test all queries thoroughly.
- Validate data constraints.
- Test performance under load.
- Automate test cases where possible.

Quick Tip: Testing ensures reliability and fewer bugs in production.

8 DOCUMENTATION

- Document database schema.
- Maintain ER diagrams.
- Document stored procedures and triggers.
- Keep change logs and version history.

Quick Tip: Good documentation saves time and avoids confusion.

GOAL

Build maintainable, secure and high-performance MySQL applications.

BEST PRACTICES REMINDER

- Plan Before You Build
- Write Clean SQL
- Monitor Regularly
- Optimize Continuously
- Collaborate Effectively
- Review & Improve Always

SUCCESS FORMULA

Good Design + Proper Indexing + Regular Maintenance = Great MySQL Project

REAL-WORLD MySQL USE CASES

Powering Scalable, Reliable & High-Performance Applications

- **SCALABLE** — Handles Growth
- **RELIABLE** — High Availability
- **FAST** — Optimized Performance
- **SECURE** — Data Protection

1 E-COMMERCE

- Store product catalogs
- Manage customers
- Process orders
- Handle payments
- Track shipping

Examples: Amazon, Flipkart, Shopify, WooCommerce

2 SCHOOL MANAGEMENT

- Manage student records
- Track attendance
- Manage fees
- Store exam results
- Teacher & class management

Examples: School ERP, Edumyra, Fedena, OpenSIS

3 HOSPITAL

- Manage patient records
- Appointment scheduling
- Billing & invoices
- Lab test records
- Pharmacy management

Examples: Practo, Hospital Management Systems, OpenEMR

4 BANKING

- Manage accounts
- Transactions & transfers
- Loan management
- Customer KYC
- Audit & compliance

Examples: Core Banking Systems, Finacle, Temenos

5 INVENTORY

- Track stock items
- Manage suppliers
- Purchase orders
- Sales tracking
- Low stock alerts

Examples: Stock Management Systems, Zoho Inventory, inFlow

6 BLOG

- Store posts & pages
- Manage categories
- User comments
- User management
- SEO & analytics data

Examples: WordPress, Ghost, Blogger, Medium Clone

7 CRM

- Manage leads
- Track interactions
- Customer profiles
- Sales pipeline
- Reports & analytics

Examples: SuiteCRM, vtiger CRM, Odoo CRM

8 ERP

- Integrate all departments
- HR & payroll
- Finance & accounting
- Supply chain management
- Reports & dashboards

Examples: Odoo, ERPNext, Dolibarr, SAP (MySQL as backend)

WHY MySQL?

- Open Source & Cost Effective
- High Performance & Scalability
- Easy to Use & Integrate
- Strong Community Support
- Trusted by Millions of Applications

MySQL is used by startups, enterprises, and developers worldwide to build powerful and data-driven applications.

MySQL QUICK REFERENCE

Common Commands • Syntax • Examples

FAST Quick Syntax | **RELIABLE** Trusted Commands | **EASY** Easy to Use | **SECURE** Data Protection

1 CREATE

Create database, table or objects.

Syntax

```
CREATE TABLE table_name (
  column1 datatype PRIMARY KEY,
  column2 datatype,
  column3 datatype
);
```

Example

```
CREATE TABLE employees (
  id INT PRIMARY KEY,
  name VARCHAR(100),
  email VARCHAR(100)
);
```

2 SELECT

Retrieve data from one or more tables.

Syntax

```
SELECT column1, column2
FROM table_name
WHERE condition;
ORDER BY column1 ASC|DESC;
```

Example

```
SELECT id, name, email
FROM employees
WHERE department = 'IT'
ORDER BY name ASC;
```

3 INSERT

Insert new records into a table.

Syntax

```
INSERT INTO table_name (column1, column2)
VALUES (value1, value2);
```

Example

```
INSERT INTO employees (id, name, email)
VALUES (1, 'Rahul Sharma', 'rahul@email.com');
```

4 UPDATE

Update existing records in a table.

Syntax

```
UPDATE table_name
SET column1 = value1, column2 = value2
WHERE condition;
```

Example

```
UPDATE employees
SET email = 'rahul@company.com'
WHERE id = 1;
```

5 DELETE

Delete records from a table.

Syntax

```
DELETE FROM table_name
WHERE condition;
```

Example

```
DELETE FROM employees
WHERE id = 1;
```

6 JOINS

Combine rows from two or more tables.

Common Joins

Join Type	Syntax	Diagram
INNER JOIN	`... FROM A INNER JOIN B ON A.id = B.id`	
LEFT JOIN	`... FROM A LEFT JOIN B ON A.id = B.id`	
RIGHT JOIN	`... FROM A RIGHT JOIN B ON A.id = B.id`	
FULL JOIN	`... FROM A LEFT JOIN B ON A.id = B.id UNION ... FROM A RIGHT JOIN B ON A.id = B.id`	

7 FUNCTIONS

Built-in functions for calculations & operations.

Common Functions

Function	Description	Example
`COUNT()`	Count rows	`SELECT COUNT(*) FROM employees;`
`SUM()`	Sum values	`SELECT SUM(salary) FROM employees;`
`AVG()`	Average value	`SELECT AVG(salary) FROM employees;`
`MAX()`	Maximum value	`SELECT MAX(salary) FROM employees;`
`MIN()`	Minimum value	`SELECT MIN(salary) FROM employees;`
`UPPER()`	Convert to upper case	`SELECT UPPER(name) FROM employees;`
`NOW()`	Current date & time	`SELECT NOW();`

8 TRANSACTIONS

Manage data with ACID properties.

Transaction Commands

Command	Description	Example
START TRANSACTION	Start a transaction	`START TRANSACTION;`
COMMIT	Save changes	`COMMIT;`
ROLLBACK	Undo changes	`ROLLBACK;`
SAVEPOINT	Set a savepoint	`SAVEPOINT sp1;`
ROLLBACK TO	Rollback to savepoint	`ROLLBACK TO sp1;`

QUICK TIPS

- Always use WHERE to avoid unintended updates/deletes.
- Use indexes to improve query performance.

BEST PRACTICES

Use Meaningful Table Names | Use Primary Keys for Every Table | Normalize Your Data | Index Important Columns | Backup Regularly | Test Queries Thoroughly | Secure Your Database

MySQL – The world's most popular open source database powering millions of applications worldwide.